TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
ĐẠI HỌC QUỐC GIA HÀ NỘI

BÁO CÁO CUỐI KỲ
MÔN HỌC: CÁC THÀNH PHẦN PHẦN MỀM

Đề tài:

DESIGN PATTERNS

Sinh viên thực hiện: HOÀNG VŨ MINH
Mã sinh viên: 20002071
Lớp: K65A5
Ngành: Khoa học dữ liệu

Hà nội, ngày 12 tháng 1 năm 2022

Contents

I Giới thiệu chung về phần tiểu luận **1**

II Giới thiệu chung về Design Patterns **2**

1 Design Pattern là gì? . . . 2
2 Lợi ích của việc sử dụng Design Pattern . . . 2
3 Các nguyên tắc của Design Pattern . . . 2
3.1 Code to an Interface . . . 2
3.2 Ưu tiên Delegation hơn Inheritance . . . 3
3.3 Tách riêng những đặc điểm thay đổi và những đặc điểm bất biến 3
4 Code minh họa về các quy tắc của Design Pattern . . . 4

III Các loại Design Pattern **9**

1 Creational Pattern . . . 9
1.1 Factory Method Pattern . . . 9
1.2 Abstract Factory Pattern . . . 17
1.3 Singleton Pattern . . . 22
1.4 Builder Pattern . . . 25
2 Structural Pattern . . . 30
2.1 Adapter Pattern . . . 30
2.2 Bridge Pattern . . . 35
2.3 Decorator Pattern . . . 39
3 Behavioral Pattern . . . 44
3.1 Strategy Pattern . . . 44
3.2 Observer Pattern . . . 46
3.3 Iterator Pattern . . . 50
3.4 Command Pattern . . . 58

IV Tài liệu tham khảo **63**

I Giới thiệu chung về phần tiểu luận

Trong phần tiểu luận này, em sẽ trình bày về 2 mục chính:

- Mục 1: Giới thiệu chung về Design Pattern, bao gồm các định nghĩa, nguyên tắc và lợi ích của việc sử dụng Design Pattern trong thiết kế chương trình.
- Mục 2: Giới thiệu về các loại Design Pattern, bao gồm việc phân loại các design pattern, với mỗi pattern, em sẽ trình bày về định nghĩa, mục đích sử dụng, sơ đồ cấu trúc, code minh họa và chúng được ứng dụng ở đâu trong thực tế.

Bài tiểu luận sẽ bao gồm một số thuật ngữ bằng tiếng Anh, em sẽ giải tích ở phần ngoặc đơn bên cạnh thuật ngữ hoặc bằng chú thích dưới chân trang. Các phần chữ được tô màu xanh da trời sẽ là những phần được hyperlink.

II Giới thiệu chung về Design Patterns

1 Design Pattern là gì?

Design Pattern là những mẫu thiết kế được đúc kết và công nhận từ nhiều chuyên gia, nhà nghiên cứu trong lĩnh vực lập trình, cung cấp cho ta giải pháp để giải quyết các vấn đề trong thiết kế chương trình một cách tối ưu.

Design Pattern sử dụng nền tảng của lập trình hướng đối tượng nên để áp dụng thành thạo Design Pattern, chúng ta cần phải nắm chắc 4 đặc trưng của lập trình hướng đối tượng: Kế thừa, Đóng gói, Trừu tượng, Đa hình.

"Mỗi pattern mô tả một vấn đề xảy ra lặp đi lặp lại, và trình bày trọng tâm của giải pháp cho vấn đề đó, theo cách mà bạn có thể dùng đi dùng lại hàng triệu lần mà không cần phải suy nghĩ". - theo kiến trúc sư Alexander Christopher

2 Lợi ích của việc sử dụng Design Pattern

Sử dụng Design Pattern giúp chúng ta tăng tốc độ phát triển phần mềm do dễ dàng nhận biết được cách giải quyết bài toán.

Giúp tránh được những vấn đề tiềm ẩn có thể gây ra lỗi sau này vì các Design Pattern là những khuôn mẫu đã qua kiểm nghiệm và sử dụng rất nhiều lần.

Giúp chúng ta dễ dàng bảo trì (mở rộng và nâng cấp) sản phẩm.

Một ưu điểm quan trọng của design pattern là giúp cho các lập trình viên dễ dàng giao tiếp và trao đổi hơn: thay vì phải trao đổi và giải thích rất lâu về hướng phát triển chương trình, chúng ta có thể chỉ ra chương trình được phát triển theo những Design Pattern nào.

3 Các nguyên tắc của Design Pattern

Nguyên tắc của một Design Pattern có thể được tóm gọn trong 3 điểm chính sau: Code to an Interface, Ưu tiên Deligation hơn Inheritance, tách riêng những đặc điểm thay đổi và đặc điểm bất biến.

3.1 Code to an Interface

Khi thiết kế chương trình, chúng ta nên cố gắng thiết kế bằng cách sử dụng tính trừu tượng: thể hiện ra bên ngoài những hành động mà chương trình có thể làm, còn việc

làm như thế nào sẽ được ẩn giấu. Chúng ta sẽ tạo ra các superclass trừu tượng trước khi thực sự thực hiện (implementation) một class nào đó.

Tuy nhiên "code to an interface" không hoàn toàn chỉ là nói đến thuật ngữ Interface trong Java mặc dù ta có thể đạt được nó thông qua việc sử dụng Interface[1] trong Java và các lớp con implements[2] lại Interface đó.

Khi chúng ta nói đến "code to an interface", điều đó ngụ ý rằng đối tượng đang sử dụng giao diện sẽ có biến là kiểu supertype, qua đó nó có thể trỏ đến bất kỳ cách triển khai nào của supertype[3] đó. Chúng ta có thể làm được điều này đó là nhờ vào khả năng của tính đa hình trong lập trình hướng đối tượng.

Một ví dụ đơn giản là thư viện List trong Java, List được thiết kế dưới dạng interface, qua đó chúng ta có thể biết được bên trong List có những tác vụ nào có thể thực thi như: get, set, remove.. nhưng phần chúng được thực thi như thế nào đã được ẩn giấu. Và với kiểu dữ liệu là supertype của mình, List có thể dễ dàng trỏ đến các lớp con của nó như ArrayList hay LinkedList, Vecto, Stack một cách dễ dàng.

Nguyên tắc này sẽ được giải thích một cách cụ thể hơn ở mục 4

Click here

3.2 Ưu tiên Delegation hơn Inheritance

Inheritance (kế thừa) là một công cụ tuyệt vời giúp ta có khả năng sử dụng lại code một cách hiệu quả, đồng bộ về mặt dữ liệu giữa các lớp cha con. Tuy nhiên việc sử dụng tính kế thừa quá nhiều sẽ gây ra cho chúng ta nhiều rắc rối về mặt thiết kế.

Delegation (ủy quyền, ủy thác) có thể hiểu là việc một đối tượng thực hiện một số hành động thông qua một đối tượng khác, tức là một đối tượng không xử lý trực tiếp một công việc mà chúng úy quyền cho một đối tượng khác để thực hiện. Chúng ta có thể hiểu Delegation giống như Composition (việc một đối tượng này chứa một đối tượng khác).

Khi chương trình có thể thiết kế theo cả 2 cách Inheritance và Delegation, chúng ta nên ưu tiên chọn cách Delegation, trừ khi các class bắt buộc phải có mối quan hệ IS A thì mới dùng Inheritance.

Nguyên tắc này sẽ được giải thích một cách cụ thể hơn ở mục 4

Click here

3.3 Tách riêng những đặc điểm thay đổi và những đặc điểm bất biến

Khi chúng ta thay đổi các đoạn code trong một chương trình, sẽ có rất nhiều đoạn code khác bị ảnh hưởng theo và nếu không cẩn thận sẽ dẫn đến chương trình bị lỗi và ta

[1] Interface: giao diện, là một lớp chỉ gồm các phương thức (trừu tượng), không chứa dữ liệu

[2] implements: triển khai, từ khóa được dùng trong Java khi ta muốn nói một lớp nào đó là lớp triển khai từ một interface, lớp đó sẽ phải thực hiện các phương thức đã khai báo từ interface đó

[3] supertype: siêu kiểu, là kiểu thực thể tổng quát của một hay nhiều kiểu con (subtype)

sẽ mất rất nhiều thời gian để tìm lỗi và sửa lại chúng. Bởi vậy, khi thiết kế một chương trình, chúng ta nên tìm cách tách các đoạn mã bất biến (không cần thay đổi) riêng ra, còn về các đoạn mã có thể phải thay đổi (bổ sung, cập nhật trong tương lai), hãy tìm cách đóng gói làm giới hạn phạm vi ảnh hưởng khi ta thực hiện thay đổi lên chúng.

Nhờ việc tuân thủ nguyên tắc này,chúng ta sẽ tránh được việc trùng code và giúp chương trình dễ dàng bảo trì và nâng cấp hơn. Nguyên tắc này rất quan trọng khi ta thực thi những dự án lớn.

Nguyên tắc này cũng góp phần giúp ta đạt được một nguyên tắc quan trọng trong lập trình đó là nguyên tắc open - closed (open for extension, close for modification): mở cho sự mở rộng, đóng cho sự thay đổi.

- "Open for extention": Chúng ta có thể mở rộng chương trình bất cứ khi nào để có thể đáp ứng được những yêu cầu mới, không phải mọi tính năng đều được đóng là cuối cùng.
- "Close for modification": Việc mở rộng của chúng ta phải không làm thay đổi và ảnh hưởng đến những phần khác trong chương trình.

4 Code minh họa về các quy tắc của Design Pattern

Phần code minh họa cho các nguyên tắc của Design pattern sẽ được trình bày tại đây Ta sẽ bắt đầu với bài toán thiết kế chương trình quản lý vườn chim, để cho đơn giản, chúng ta chỉ cần quan tâm đến phương thức fly() (kiểu bay) của các loài chim đó. Ban đầu, chúng ta chỉ có 2 loài chim trong vườn đó là SkyBird và MountainBird, với khả năng bay thấp (chỉ cách mặt đất 2km).

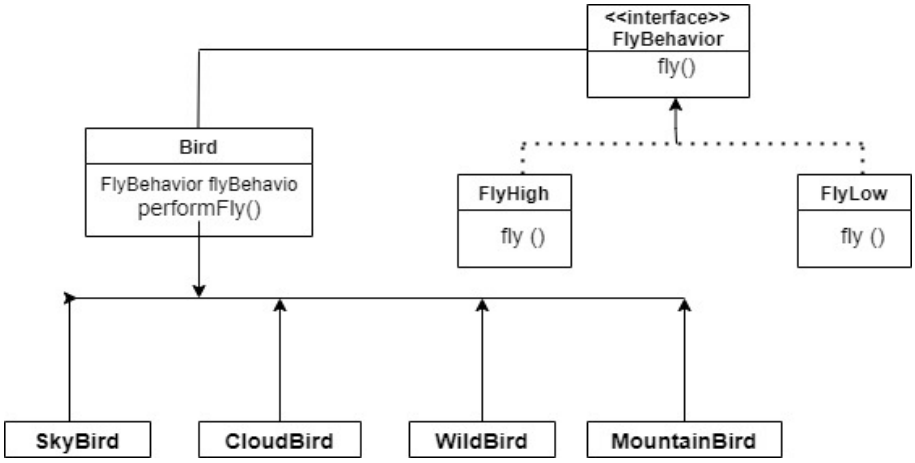
Với cách làm thông thường, đầu tiên chúng ta sẽ tạo ra một lớp chung là Bird, sau đó tạo lớp MountainBird và SkyBird là các lớp con của Bird. Vì 2 lớp con này có chung cách bay, nên ta chỉ cần viết phương thức fly() một lần ở lớp Bird, tính kế thừa sẽ khiến cho cả 2 lớp con đều có được phương thức này.

```
    // Bird.java
public class Bird {
   String name;

   public void fly() {
      System.out.println("Fly 2km from ground");
   }
}
    // MountainBird.java
public class MountainBird {
```

```
}
    // SkyBird.java
public class SkyBird {

}
```

Nếu bài toán chỉ dừng lại ở đó thì cách xử lý này cũng tạm ổn, tuy nhiên sau đó, vườn chim có thêm 2 loài chim mới là loài CloudBird và WildBird với khả năng bay cao (10km so với mặt đất).

Nếu chúng ta tiếp tục giải quyết theo tính kế thừa, chúng ta sẽ cần phải thay đổi cấu trúc chương trình và sửa lại code rất nhiều vì phải tạo thêm 2 lớp FlyHigh và FlyLow kế thừa Bird, sau đó MoutainBird và SkyBird kế thừa lớp FlyLow, CloudBird và WildBird kế thừa lớp FlyHigh, các đoạn code đã viết trong lớp Bird cũng phải thay đổi lại hết. Ở ví dụ này, chúng ta đang đặt ra bài toán chỉ viết phương thức fly() nên mọi chuyện vẫn đơn giản, nếu có thêm phương thức như eat(), sẽ có những loài chim khác kiểu bay cùng kiểu ăn hoặc cùng kiểu bay khác kiểu ăn. **Việc cố gắng tiếp tục sử dụng Inheritance[4] lúc này sẽ khiến chương trình trở nên phức tạp hơn rất nhiều.** Thay vào đó, bằng việc vận dụng Delegate (ủy quyền) và code to an interface, chúng ta sẽ dễ dàng tạo ra một chương trình hợp lý, dễ bảo trì và mở rộng, chương trình sau khi được thay đổi bằng cách vận dụng chúng sẽ có sơ đồ như sau:



Đầu tiên, ta sẽ tạo một lớp Bird là lớp trừu tượng, các lớp con SkyBird, CloudBird, WildBird, MountainBird được extends từ lớp Bird. Điều này giúp cho chúng ta có được sự đồng bộ dữ liệu giữa lớp Bird và các lớp con của nó, và với tính đa hình về kiểu dữ liệu (datatype polymorphism), chúng ta có thể dễ dàng quản lý tất cả các loài chim chỉ với 1 List kiểu Bird.

Tiếp đến là vấn đề hành vi bay. Thay vì sử dụng thừa kế như trên, chúng ta sẽ sử dụng Delegation (ủy quyền) bằng cách tạo một biến flyBehaivior có kiểu dữ liệu là

[4]kế thừa: sự liên quan giữa 2 class với nhau theo mối quan hệ IS - A

FlyBehavior trong lớp Bird (compositon). Lúc này mọi hành vi bay của các loài chim sẽ được thực hiện thông qua FlyBehavior.

Trước khi thiết kế từng lớp hành vi bay cụ thể như FlyHigh hay FlyLow, chúng ta sẽ tạo interface FlyBehavior, Flybahavior sẽ là supertype của tất cả các hành vi bay này, sau đó cho các lớp FLylow và FlyHigh implements từ FlyBehavior. Bằng cách thiết kế này, một chú chim sẽ có khả năng thực hiện bất kỳ khả năng bay nào có trong FLyBehavior, tùy từng loại chim ta sẽ đặt cho chúng hành vi bay khác nhau. Nếu ta sử dụng luôn là FlyHigh thì bắt buộc các loài chim đều chỉ có thể bay cao. Việc sử dụng kỹ thuật này khiến chương trình trở nên linh hoạt hơn rất nhiều, đây chính là kỹ thuật **"code to an interface".**

Bằng cách dùng "Delegation" và "Code to an interface", chúng ta cũng có thể dễ dàng mở rộng chương trình này một cách dễ dàng: ví dụ khi cần thêm EatBehavior, ta cũng làm tương tự như FlyBehavior.

Code minh họa: Phần FLybehavior

```
    // FlyBehavior.java
public interface FlyBehavior {
   void fly();
}
    // FlyHigh.java
public class FlyHigh implements FlyBehavior {
   public void fly() {
      System.out.println("Fly over 10 km from the ground");
   }
}
    //FlyLow.java
public class FlyLow implements FlyBehavior {
   public void fly() {
      System.out.println("Fly 2km from the ground");
   }
}
```

Code minh họa: Phần Bird. Ở đây ta thấy ở lớp Bird có thêm phương thức setFlyBehavior() cho phép ta có thể thay đổi phương thức bay ở ngay lúc runtime. Đây cũng là một phần sức mạnh của việc sử dụng Delegate (ủy quyền). Các hành vi bay như FlyLow() hay FlyHigh() sẽ được ta đặt cho từng loài chim thông qua hàm dựng (constructor) của chúng.

```
   // Bird.java
abstract public class Bird {
   protected String name;
```

```
    protected FlyBehavior flyBehavior;

    public void setFlyBehavior(FlyBehavior fb) {
        flyBehavior = fb;
    }

    public Bird() {
    }

    public void performFly() {

        flyBehavior.fly();
    }
}
    // SkyBird.java
public class SkyBird extends Bird {
    public SkyBird() {
        flyBehavior = new FlyLow();
    }
}
    // MountainBird.java
public class MountainBird extends Bird {
    public MountainBird() {
        flyBehavior = new FlyLow();
    }
}
    // CloudBird.java
public class CloudBird extends Bird {
    public CloudBird() {
        flyBehavior = new FlyHigh();
    }
}
    // Wildbird.java
public class WildBird extends Bird {
    public WildBird() {
        flyBehavior = new FlyHigh();
    }
}
```

Để kiểm chứng tính chính xác của chương trình, ta sẽ cùng thử qua một lớp test nhỏ:

```
    // Test.java
```

```
import java.util.*;

public class Test {
   public static void main(String[] args) {
      Bird cloudBird = new CloudBird();
      Bird skyBird = new SkyBird();
      Bird mountainBird = new MountainBird();
      Bird wildBird = new WildBird();

      List<Bird> birds = new ArrayList<>();
      birds.add(wildBird);
      birds.add(skyBird);
      birds.add(mountainBird);
      birds.add(cloudBird);

        // Change the fly behavior:
        // skyBird.setFlybehavior(new FlyHigh());

      System.out.println("This is a cloud bird: ");
      cloudBird.performFly();
   }
}
```

Sức mạnh của "Code to an interface" cho phép ta dễ dàng quản lý tất cả các loài chim chỉ thông qua một List<Bird> duy nhất.

Output:

```
This is a cloud bird
Fly over 10 km from the ground
```

Như trên lớp Test ta có thể thấy, các loài chim khác nhau có thể được quản lý chung thông qua List<Bird> và khi performFly() của CloudBird cho ra kết quả "Fly over 10km from the ground" vì Cloud Bird là loài chim bay cao.

Ví dụ trên chính là cách thiết kế một chương trình theo một Design Pattern, cụ thể ở đây là Strategy Pattern thuộc nhóm Behavioral. Để xem thêm về Strategy Pattern, bấm vào Strategy Pattern.

III Các loại Design Pattern

1 Creational Pattern

Là nhóm Design Pattern liên quan đến cơ chế tạo đối tượng, giúp ta tạo được đối tượng phù hợp với tình huống bài toán đặt ra. Các Creational Pattern có thể kể đến đó là: Singleton, Factory, Builder, Prototype...

1.1 Factory Method Pattern

Factory Method Pattern là gì? Factory Method Pattern (nhà máy) là một pattern thuộc nhóm khởi tạo, nó định nghĩa một interface dùng để khởi tạo đối tượng nhưng cho phép các lớp con quyết định lớp nào được khởi tạo. Nói cách khác, Factory Pattern quản lý và trả về các đối tượng theo yêu cầu, điều này giúp cho việc khởi tạo đối tượng trở nên linh hoạt hơn.

Mục đích của việc sử dụng Factory Pattern: Factory Method Pattern được sử dụng khi ta muốn khởi tạo đối tượng theo một cách mới, pattern này giúp che giấu logic của việc khởi tạo đối tượng. Nó cũng giúp cho việc khởi tạo đối tượng trở nên độc lập và có thể mở rộng dễ dàng mà không làm ảnh hưởng đến các đoạn code khác.
Trước khi đi vào phân tích Factory Method Pattern, chúng ta hãy cùng phân tích một pattern khác đó là Simple Factory Pattern (tuy nhiên người ta không coi nó là một pattern nữa nên trong các tài liệu chỉ còn chia làm Factory Method Pattern và Abstract Factory Pattern). Thuật ngữ Factory Pattern là để chỉ Factory Method Pattern.

Khi ta khởi tạo đối tượng theo các thông thường: Giả sử ta viết một chương trình về cửa hàng Pizza, ta đã có sẵn các lớp Pizza và các loại Pizza như ChesePizza... Lúc này ta cần viết một lớp là PizzaStore để có thể gọi Pizza theo đúng loại yêu cầu. Thông thường chương trình sẽ được triển khai:

```
    // PizzaStore.java
public class PizzaStore {
   Pizza orderPizza(String type) {
      Pizza pizza;
      if (type.equals("cheese")) {
         pizza = new CheesePizza();
      } else if (type.equals("greek")) {
         pizza = new GreekPizza();
      } else if (type.equals("pepperoni")) {
         pizza = new PepperoniPizza();
      }
```

```
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

Nhờ nguyên tắc "code to interface", một biến kiểu Pizza có thể dễ dàng trỏ tới các lớp con của nó, đây là một ví dụ áp dụng rất tốt nguyên tắc này. Tuy nhiên như ta thấy, nếu để việc khởi tạo đối tượng ngay trong lớp PizzaStore, đặc biệt là trong phương thức quan trọng orderPizza(), mỗi khi có thêm loại pizza mới, ta phải mở ra và sửa lại code ngay tại đây, điều này có thể dẫn đến việc thay đổi các đoạn code khác (vi phạm open - closed).

Giải pháp: Chúng ta có thể giải quyết vấn đề này bằng cách chuyển việc khởi tạo sang một lớp hoàn toàn khác. Chúng ta sẽ đóng gói việc khởi tạo lại thành một class riêng, và class này chính là Factory. Class này chỉ có duy nhất 1 nhiệm vụ đó chính là khởi tạo đối tượng theo yêu cầu.

```
    // SimplePizzaFactory.java
public class SimplePizzaFactory {
    public Pizza createPizza(String type) {
        if (type.equals("cheese")) {
            return new CheesePizza();
        } else if (type.equals("greek")) {
            return new GreekPizza();
        } else if (type.equals("pepperoni")) {
            return new PepperoniPizza();
        }
    }
}
```

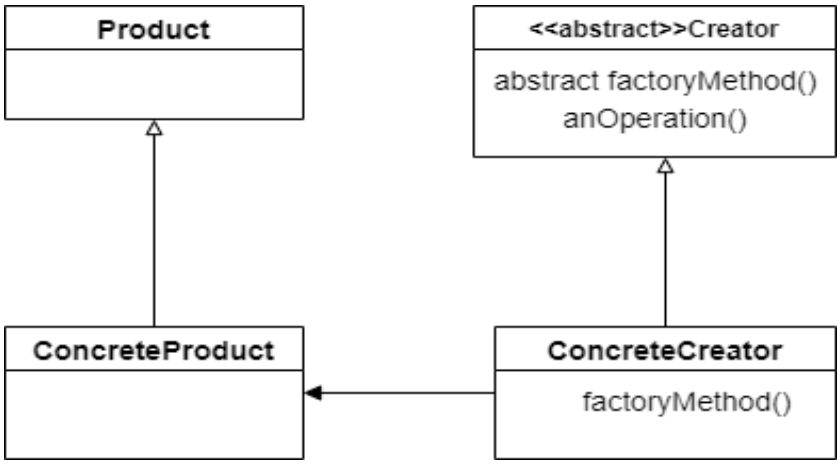
Bằng cách làm này, việc tạo đối tượng đã được ủy quyền sang các Factory. Mỗi khi ta muốn thêm một loại Pizza mới, ta chỉ cần sửa trong lớp SimplePizzaFactory, điều này sẽ không làm ảnh hưởng tới các phần cố định khác của chương trình. Đó chính là hiệu quả của việc sử dụng Factory Pattern. Tuy nhiên như chúng ta thấy, chương trình chưa được linh hoạt cho lắm:

- Giả dụ, vì PizzaStore quá thành công nên chúng ta có thể mở rộng nó ra thành 2 chi nhánh đó là NYPizzaStore và ChicagoPizzaStore.
- Nếu tư duy theo cách sử dụng Simple Factory như trên, chúng ta sẽ tạo ra NYFactory và ChicagoFactory để tạo Pizza cho 2 chi nhánh nhỏ này.

- Tuy nhiên, nếu làm như vậy, chúng ta đã vô tình mặc định rằng ở các chi nhánh, những chiếc bánh Pizza đều được tạo ra như nhau, mà trong thực tế, sự khác nhau trong các kỹ thuật chế tạo ở các chi nhánh đã làm nên đặc trưng của các chi nhánh này.
- Chính vì vậy, chúng ta cần tìm một cách thiết kế khác sao cho vừa có thể đặc trưng hóa cách tạo Pizza của từng vùng miền mà vẫn đảm bảo tính linh hoạt, chặt chẽ của chương trình.

Để thực hiện điều này, chúng ta sẽ sử dụng Factory Method Pattern, nói một cách đơn giản, chúng ta sẽ đưa nhà máy vào trong phương thức và thực hiện ở các lớp con.

Sơ đồ thiết kế Factory Method Pattern Factory Method Pattern được thiết kế theo sơ đồ sau:



Ta có thể hiểu sơ đồ này như sau:

- Lớp Creator được thiết kế là một lớp trừu tượng, điều này vừa để đảm bảo nguyên tắc "code to an interface" vừa nhằm tạo ra Factory Method - điểm quan trọng nhất trong pattern này. Các phương thức khác có trong Creator đều được thực thi bình thường, ngoại trừ factoryMethod() do ta khai báo nó là phương thức trừu tượng.
- Phương thức factoryMethod() chính là phương thức được sử dụng nhằm tạo đối tượng thay cho các Simple Factory phía trên, bằng cách ghi đè lại factoryMethod() ở lớp Creator, các lớp con ConcreteCreator có thể tự định nghĩa lại phương thức này, quyết định kiểu đối tượng mà chúng muốn tạo. factoryMethod() có thể có tham số hoặc không, tùy thuộc vào cách thiết kế của chúng ta.
- ConcreteCreator sẽ chịu trách nghiệm tạo ra ConcreteProduct, ConcreteProduct được implements từ lớp trừu tượng Product, điều này giúp cho ta đạt được nguyên tắc "code to interface", dễ dàng quản lý các sản phẩm hơn.

Code minh họa PizzaStore theo Factory Method Pattern Bây giờ, chúng ta sẽ vận dụng sơ đồ cấu trúc bên trên để thiết kế lại của hàng Pizza nhượng quyền ở cả New York và Chicago. Ứng với sơ đồ ta sẽ có:

- Creator sẽ ứng với PizzaStore, factoryMethod() lúc này sẽ tương ứng với phương thức createPizza().
- Các ConcreteCreator sẽ lần lượt là NYPizzaStore và ChicagoPizzaStore.
- Product sẽ là lớp Pizza chứa một số phương thức để chuẩn bị một chiếc Pizza, trong khi đó các lớp ConcreteProduct sẽ là các loại Pizza như NYCheesePizza, NYClamPizza, ChicagoCheesePizza, ChicagoClamPizza.

Code lớp PizzaStore trừu tượng (code to interface): Theo như sơ đồ, lúc này chúng ta sẽ đưa phương thức createPizza() quay trở lại lớp Pizza nhưng dưới dạng một phương thức trừu tượng. Trong PizzaStore còn có thêm phương thức orderPizza() dùng để gọi pizza theo đúng loại ứng với đối số mà ta truyền vào.

```
    // PizzaStore.java
public class PizzaStore {
   SimplePizzaFactory factory;

   public PizzaStore(SimplePizzaFactory factory) {
      this.factory = factory;
   }

   public Pizza orderPizza(String type) {
      Pizza pizza;

      pizza = factory.createPizza(type);

      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();

      return pizza;
   }
}
```

Tiếp đến, chúng ta tạo các lớp NYPizzaStore và ChicagoPizzaStore implements từ PizzaStore. Do ràng buộc về tính trừu tượng, các lớp này sẽ phải thực hiện ghi đè lên phương thức createPizza() để tạo ra các loại Pizza đặc trưng cho từng vùng miền. Ta không ghi đè orderPizza() vì muốn tất cả các chi nhánh đều có quy trình gọi pizza giống nhau.

Khi orderPizza() được gọi, nó sẽ gọi đến createPizza() để tạo ra pizza, tuy nhiên phương thức orderPizza() không thể quyết định loại pizza nào sẽ được tạo (do createPizza() là một phương thức trừu tượng), chính các lớp NYPizzaStore và ChicagoPizzaStore sẽ xác định loại pizza nào được làm, tùy vào lựa chọn của người sử dụng.

```
    // NYPizzaStore.java
public class NYPizzaStore extends PizzaStore {

   Pizza createPizza(String item) {
      if (item.equals("cheese")) {
         return new NYStyleCheesePizza();
      } else if (item.equals("clam")) {
         return new NYStyleClamPizza();
      } else {
         return null;
      }
   }
}
    // ChicagoPizzaStore.java
public class ChicagoPizzaStore extends PizzaStore {

   Pizza createPizza(String item) {
      if (item.equals("cheese")) {
         return new ChicagoStyleCheesePizza();

      } else if (item.equals("clam")) {
         return new ChicagoStyleClamPizza();
      } else {
         return null;
      }
   }
}
```

Như vậy chúng ta đã thiết kế xong một phần rất quan trong đó chính là các cửa hàng pizza nhượng quyền, bây giờ chúng ta chỉ cần tạo ra các loại Pizza nữa thôi. Lớp Pizza sẽ gồm các phương thức để chuẩn bị pizza như prepare(), bake(), cut(), box(), chúng ta sẽ gọi đến các phương thức này trong orderPizza() ở PizzaStore để chúng có thể thực hiện ở tất cả các cửa hàng.

Lớp này cũng được thiết kế là một lớp trừu tượng vì trong thực tế không tồn tại một loại Pizza chung nào cả.

```
    // Pizza.java
```

```
import java.util.ArrayList;

public abstract class Pizza {
   String name;
   String dough;
   String sauce;
   ArrayList<String> toppings = new ArrayList<String>();

   void prepare() {
      System.out.println("Prepare " + name);
      System.out.println("Tossing dough...");
      System.out.println("Adding sauce...");
      System.out.println("Adding toppings: ");
      for (String topping : toppings) {
         System.out.println(" " + topping);
      }
   }

   void bake() {
      System.out.println("Bake for 25 minutes at 350");
   }

   void cut() {
      System.out.println("Cut the pizza into diagonal slices");
   }

   void box() {
      System.out.println("Place pizza in official PizzaStore box");
   }

   public String getName() {
      return name;
   }

   public String toString() {
      StringBuffer display = new StringBuffer();
      display.append("---- " + name + " ----\n");
      display.append(dough + "\n");
      display.append(sauce + "\n");
      for (String topping : toppings) {
         display.append(topping + "\n");
      }
```

```
        return display.toString();
    }
}
```

Một số loại pizza được implements từ lớp Pizza, điều này khiến cho Pizza có thể trỏ đến tất cả các kiểu con của nó:

```
    // ChicagoStyleCheesePizza.java
public class ChicagoStyleCheesePizza extends Pizza {

    public ChicagoStyleCheesePizza() {
        name = "Chicago Style Deep Dish Cheese Pizza";
        dough = "Extra Thick Crust Dough";
        sauce = "Plum Tomato Sauce";

        toppings.add("Shredded Mozzarella Cheese");
    }

    void cut() {
        System.out.println("Cutting the pizza into square slices");
    }
}

    // ChicagoStyleClamPizza.java
public class ChicagoStyleClamPizza extends Pizza {

    public ChicagoStyleClamPizza() {
        name = "Chicago Style Clam Pizza";
        dough = "Extra Thick Crust Dough";
        sauce = "Plum Tomato Sauce";

        toppings.add("Shredded Mozzarella Cheese");
        toppings.add("Frozen Clams from Chesapeake Bay");
    }

    void cut() {
        System.out.println("Cutting the pizza into square slices");
    }
}

    // NYStyleCheesePizza.java
public class NYStyleCheesePizza extends Pizza {
```

```
    public NYStyleCheesePizza() {
        name = "NY Style Sauce and Cheese Pizza";
        dough = "Thin Crust Dough";
        sauce = "Marinara Sauce";

        toppings.add("Grated Reggiano Cheese");
    }
}

    // NYStyleClamPizza.java
public class NYStyleClamPizza extends Pizza {

    public NYStyleClamPizza() {
        name = "NY Style Clam Pizza";
        dough = "Thin Crust Dough";
        sauce = "Marinara Sauce";

        toppings.add("Grated Reggiano Cheese");
        toppings.add("Fresh Clams from Long Island Sound");
    }
}
```

Lớp test: để mô phỏng việc chọn cửa hàng nào tạo Pizza, ta chỉ cần khởi tạo PizzaStore đó.

```
    // PizzaTestDrive.java
public class PizzaTestDrive {

  public static void main(String[] args) {
    PizzaStore nyStore = new NYPizzaStore();
    PizzaStore chicagoStore = new ChicagoPizzaStore();

    Pizza pizza = nyStore.orderPizza("cheese");
    System.out.println("Ethan ordered a " + pizza.getName() + "\n");

    pizza = chicagoStore.orderPizza("cheese");
    System.out.println("Joel ordered a " + pizza.getName() + "\n");

    pizza = nyStore.orderPizza("clam");
    System.out.println("Ethan ordered a " + pizza.getName() + "\n");
```

```
    pizza = chicagoStore.orderPizza("clam");
    System.out.println("Joel ordered a " + pizza.getName() + "\n");
  }
}
```

Vừa rồi là phần code minh họa về Factory Method Pattern dựa trên một bài toán thực tế là cửa hàng pizza nhượng quyền.

Tuy nhiên, ta thấy rằng việc sử dụng Factory Method Pattern rất hữu ích khi ta chỉ muốn tạo ra một loại đối tượng (cụ thể trong bài toán này là tạo ra Pizza). Nhưng trong trường hợp cần tạo ra nhiều hơn 1 đối tượng, Factory Method Pattern sẽ rất khó để thực hiện được điều này. Khi đó, chúng ta cần một pattern khác cũng thuộc nhóm Factory, có tên Abstract Factory.

Một vài chương trình sử dụng Factory Method Pattern trong thực tế Factory method pattern được tìm thấy trong một số thư viện của Java như java.util.Calendar, NumberFormat, ResourceBundle...
Bấm vào đây để trở lại mục lục.

1.2 Abstract Factory Pattern

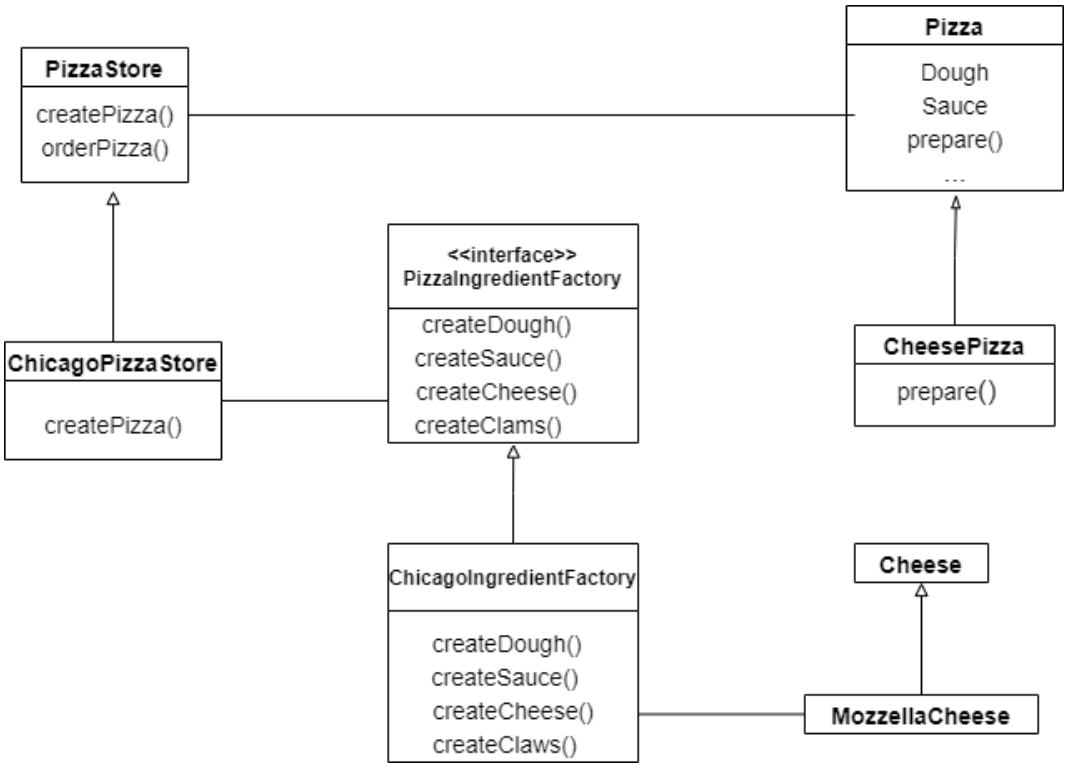
Phần trình bày về Abstract Factory Pattern dưới đây sẽ liên quan rất nhiều đến bài toán PizzaStore đã đặt ra khi trình bày về Factory Method Pattern. Xem lại Factory Method Pattern tại Factory Method Pattern.

Abstract Factory Pattern là gì? Abstract Factory Pattern là một pattern kiểu khởi tạo cung cấp một interface có thể tạo ra một họ các đối tượng liên quan hoặc phụ thuộc vào nhau. Abstract Factory đóng gói một nhóm các lớp có vai trò sản xuất (các Factory), mỗi Factory sẽ tạo ra đối tượng giống như cách hoạt động của Simple Factory.

Mục đích sử dụng Abstract Factory Pattern: Abstract Factory Pattern có mục đích sử dụng giống như Factory Method, tuy nhiên chúng ta sử dụng Abstract Factory Pattern khi cần tạo ra nhiều đối tượng hơn.

Tiếp tục với bài toán PizzaStore: Trong bài toán PizzaStore ở phần Factory Method bên trên, ban đầu chúng ta để các phụ gia như dough, sauce chỉ đơn giản là kiểu String. Tuy nhiên trên thực tế, ở mỗi vùng cần các loại thành phần phụ gia khác nhau, mỗi loại pizza khác nhau lại có các thành phần phụ gia khác nhau. Lúc này ta sẽ biến chúng trở thành các đối tượng. Và phụ gia thì sẽ có nhiều loại nên chúng ta sẽ không thể chỉ sử dụng Factory Method như trên được. Để xử lý bài toán này, chúng ta sẽ cải tiến lại

PizzaStore theo sơ đồ như sau (sơ đồ đã được giản lược để dễ dàng minh họa sự khác biệt):



Một vài điểm thay đổi đáng chú ý:

- Ta sẽ tạo ra một interface PizzaIngredientFactory khai báo các phương thức để tạo ra các loại nguyên liệu. Để các loại nguyên liệu riêng cho từng miền, các nhà máy ở từng địa phương sẽ thực hiện implements interface này và viết lại các hàm create().
- Do nguyên liệu ở các vùng đã khác nhau nên ở lớp Pizza, phương thức prepare() sẽ được chuyển thành abstract method, các lớp con lúc này sẽ override lại phương thức để tạo ra các bước chuẩn bị khác nhau giữa các vùng.
- Đặc biệt, chúng ta không cần chia thành NY Style hay Chicago Style nữa. Việc phân biệt style pizza giữa các vùng giờ đây sẽ được dựa vào phụ gia của chúng được lấy từ nhà máy nào. Đây chính là điểm khác quan trọng trong thiết kế chương trình, giờ đây nhà máy sẽ quyết định sự khác biệt.
- Chúng ta cũng cần thiết kế thêm các lớp trừu tượng Dough, Sauce,...và các lớp con của chúng.

Ta có thể đạt được thiết kế của sơ đồ này bằng việc kết hợp nhuần nhuyễn kỹ thuật "code to interface" và delegate (ủy quyền). Chúng giúp cho chương trình trở nên rất linh hoạt, dễ mở rộng và sửa đổi.

Phần code minh họa: Phần code dưới đây đã được giản lược đi nhiều, tập trung vào việc thể hiện sự thay đổi ở các lớp chính.

Code lớp PizzaIngredientFactory: code to interface, cho phép ta linh hoạt sử dụng các nhà máy bên dưới. Lớp này hiển thị các phương thức tạo phụ gia.

```
    // PizzaIngredientFactory.java
public interface PizzaIngredientFactory {
   public Dough createDough();

   public Sauce createSauce();

   public Cheese createCheese();

   public Clams createClams();

   ...
}
```

Tiếp theo là lớp ChicagoIngredientFactory implements từ PizzaIngredientFactory. Lớp này sẽ ghi đè các phương thức tạo phụ gia để tạo ra phụ gia riêng cho vùng Chicago (tương tự với vùng NY).

```
    // ChacagoIngredidentFactory.java
public class ChicagoPizzaIngredientFactory implements
    PizzaIngredientFactory {
   public Dough createDough() {
      return new ThickCrustDough();
   }

   public Sauce createSauce() {
      return new PlumTomatoSauce();
   }

   public Cheese createCheese() {
      return new MozzarellaCheese();
   }
   ...
}
```

Vậy là đã xong thiết kế phần nhà máy. Tiếp theo chúng ta sẽ thiết kế các lớp Pizza sao cho chúng có thể sử dụng các nhà máy phụ gia để tạo ra các style pizza khác nhau riêng biệt.

Đầu tiên là lớp Pizza trừu tượng với một phương thức trừu tượng là prepare(). Sau đó lớp con CheesePizza sẽ ghi đè lại phương thức prepare() đó.

```
    // Pizza.java
public abstract class Pizza {
   String name;
   Dough dough;
   Sauce sauce;
   Cheese cheese;
   Clams clam;

   abstract void prepare();
   ...
}
    // CheesePizza.java
public class CheesePizza extends Pizza {
   PizzaIngredientFactory ingredientFactory;

   public CheesePizza(PizzaIngredientFactory ingredientFactory) {
      this.ingredientFactory = ingredientFactory;
   }

   void prepare() {
      System.out.println("Preparing " + name);
      dough = ingredientFactory.createDough();
      sauce = ingredientFactory.createSauce();
      cheese = ingredientFactory.createCheese();
   }
}
```

Đây chính là cách mà các Factory được đưa vào sử dụng để tạo ra những chiếc pizza. Bằng kỹ thuật ủy quyền (delegate), các hành động tạo phụ gia đã được các lớp pizza cụ thể ủy quyền sang cho các nhà máy thực hiện. Tùy theo chúng ta chọn nhà máy ở vùng nào (Chicago hoặc NY), nguyên liệu của Pizza sẽ được tạo ra ứng với nguyên liệu của nhà máy ở từng vùng, chiếc pizza lúc này sẽ mang style của vùng tương ứng đó (chọn bằng cách truyền Factory vào constructor).

Bằng cách này, việc chia ra thành NYCheeseStyle và ChicagoCheeseStyle đã không còn cần thiết. Cuối cùng, chúng ta sẽ thay đổi một chút ở lớp con ChicagoPizzaStore

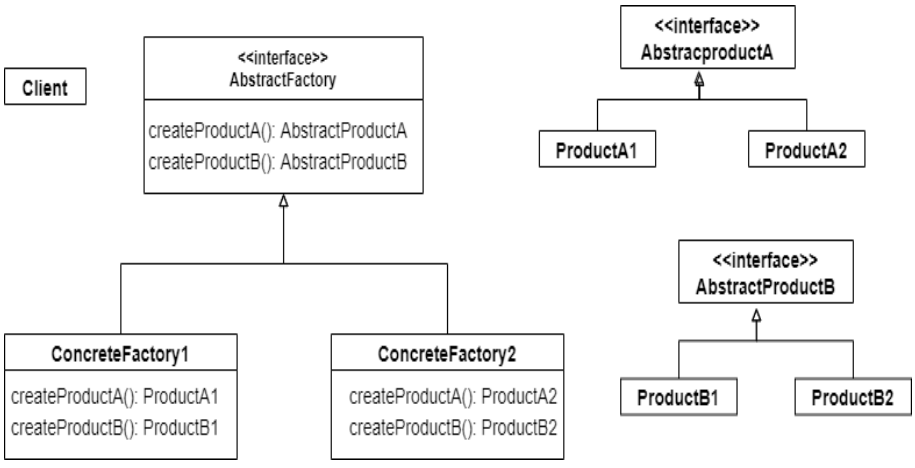
```
    // ChicagoPizzaStore.java
public class ChicagoPizzaStore extends PizzaStore {
   protected Pizza createPizza(String item) {
```

```
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory = new
            ChicagoPizzaIngredientFactory();
        if (item.equals("cheese")) {
           pizza = new CheesePizza(ingredientFactory);
           pizza.setName("Chicago Style Cheese Pizza");
           } else if (item.equals("clam")) {
           pizza = new ClamPizza(ingredientFactory);
           pizza.setName("Chicago Style Clam Pizza");
           }
           . . .
    } return pizza;
}
```

Như ta thấy, ta sẽ tạo ra các Pizza bằng việc sử dụng Constructor (hàm dựng) của các lớp Pizza cụ thể. Ta sẽ truyền vào hàm dựng các Factory và dùng setName() để thay đổi tên của chiếc Pizza, như vậy ta có thể tạo ra pizza mang style của bất kỳ vùng nào.

Đó là ví dụ về bài toán thiết kế cửa hàng Pizza bằng Abstract Factory Pattern (có cả sự kết hợp của Factory method trong phương thức createPizza()).

Sơ đồ tổng quát hóa của Abstract Factory Pattern: Ta sẽ tổng quát hóa sơ đồ của PizzaStore thành sơ đồ của Abstract Factory



Abstract Factory sử dụng rất hiệu quả khi ta cần tạo ra nhiều đối tượng. Cách triển khai của sơ đồ này giống với phần giải thích về chương trình Pizza ở trên.

Một vài chương trình thực tế sử dụng Abstract Factory Pattern Abstract Factory Pattern thường được sử dụng cùng với Factory Method pattern (ví dụ như chương trình cửa hàng pizza ở trên). trong JDK, chúng ta có thể tìm thấy pattern này trong newInstance() của:

- javax.xml.parsers.DocumentBuilderFactory;
- javax.xml.transform.TransfomerFactory;

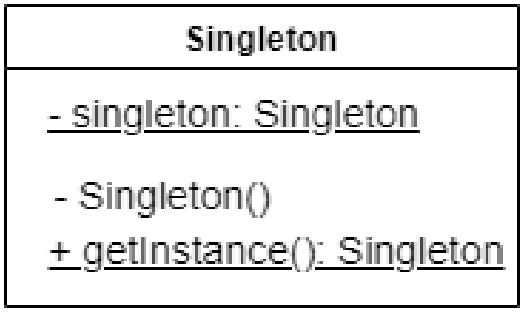
Bấm vào đây để trở lại mục lục.

1.3 Singleton Pattern

Singleton Pattern là gì? Singleton Pattern là một trong những pattern được sử dụng rộng rãi và phổ biến, thuộc nhóm khởi tạo. Pattern này cho phép ta chỉ tạo được một đối tượng duy nhất từ một lớp. Nó còn cung cấp cho ta một phương thức để có thể truy xuất được đối tượng duy nhất đó ở mọi nơi trong chương trình.

Mục đích sử dụng: Singleton Pattern được sử dụng khi ta muốn tạo ra một đối tượng duy nhất từ một lớp. Ngoài ra, chúng ta còn có thể sử dụng Singleton Pattern như một cách để thay cho biến toàn cục (global variable: biến có thể truy cập ở bất cứ đâu trong chương trình). Việc sử dụng nhiều biến toàn cục trong một chương trình là một điều không tốt vì ta sẽ rất khó kiểm soát và thay thế chúng, bởi vậy Singleton Pattern là một giải pháp thay thế tuyệt vời.

Sơ đồ cấu trúc Singleton Pattern: Sơ đồ cầu trúc của Singleton Pattern có thể thể hiện đơn giản như sau:



Các thành phần của sơ đồ

- Vì Singleton Pattern liên quan đến việc hạn chế tạo đối tượng (chỉ được tạo một đối tượng từ một lớp), nên lúc này ta sẽ chuyển hàm Constructor của lớp Singleton thành mức truy cập private[5].
- phương thức getInstance() sẽ được thiết kế là một phương thức tĩnh (static method), phương thức này cho phép ta tạp ra một đối tượng từ một lớp nếu nó chưa tồn tại ở bất kỳ đâu trong chương trình.

[5]private: mức truy cập chỉ cho phép ta nhìn thấy trong lớp hiện tại

- Cuối cùng là class data member Singleton singleton, được đặt là static nhằm trở thành thuộc tính chung của các đối tượng.

Chúng ta sẽ giải thích kỹ hơn sơ đồ này thông qua phần code minh họa.

Code minh họa Singleton Pattern Dưới đây là phần code class Ball (bóng) thông qua việc sử dụng Singleton Pattern để khiến nó chỉ có thể tạo được một đối tượng ball duy nhất:

```
    // Ball.java
public class Ball {
   private static Ball ball;
   private String color;

   private Ball(String color) {
      this.color = color;
   }

   public void bounce() {
      System.out.println("Boing!");
   }

   public void display() {
      ball.bounce();
      System.out.println(color);
   }

   public static Ball getInstance(String color) {
      if (ball == null) {
         ball = new Ball(color);
      }
      return ball;
   }
}
```

Đây là cách ta sử dụng Singleton Pattern để tạo ra một đối tượng ball duy nhất từ lớp Ball:

- Như ta thấy trên code, để hạn chế việc tạo đối tượng bằng Constructor, ta sẽ chuyển nó về mức truy cập private.
- Tuy nhiên, nếu chỉ chuyển Constructor thành mức truy cập private, ta thậm chí không thể tạo được đối tượng nào. Bởi vậy, tận dụng phạm vi hoạt động của

mức truy cập private (truy cập tự do trong cùng lớp), ta sẽ tạo ra phương thức getInstance(String color) để tạo ra đối tượng với màu truyền vào.

- phương thức này sẽ cho phép truyền "color" vào để tạo ra trái bóng mong muốn. Nếu "ball == null" (chưa có đối tượng nào), ta sẽ tạo ra một đối tượng mới, còn nếu có đối tượng rồi, ta sẽ trả lại đối tượng hiện tại bằng lệnh "return ball".
- Tuy nhiên khi so sánh "ball == null", ball cần phải tham chiếu đến đối tượng mà mình muốn tạo ra duy nhất. Do đó, ta cần thêm từ khóa static vào thuộc tính "ball" để nó có thể trở thành thuộc tính chung cho tất cả đối tượng của lớp.
- Phương thức này phải là static method[6] vì static method sẽ thuộc về class và có trước đối tượng, việc này sẽ đảm bảo phương thức này có trước đối tượng chúng ta cần tạo, nên ta mới có thể dùng nó để tạo đối tượng. Static method cũng có thể được gọi trực tiếp bằng tên class.
- Phương thức này cũng được thiết kế theo cơ chế "lazy instantiation", nghĩa là khi nào gọi đến thì mới thực hiện, đối tượng được tạo ra chỉ khi ta getInstance().

Lớp Test đơn giản: Ta sẽ sử dụng hàm display() để kiểm chứng xem có phải chỉ duy nhất 1 đối tượng được tạo ra hay không.

```
    // TestBall.java
public class TestBall {
   public static void main(String[] args) {
      Ball b1 = Ball.getInstance("Red");
      b1.display();
      Ball b2 = Ball.getInstance("Blue");
      b2.display();
   }
}
```

Output:

```
Boing!
Red
Boing!
Red
```

Ta thấy, mặc dù đối tượng b2 được khởi tạo với màu là "Blue" nhưng kết quả của hàm display() vẫn là "Red", điều đó chứng tỏ không có thêm đối tượng mới nào cả.

[6]static method: phương thức tĩnh, là các phương thức thuộc lớp chứ không thuộc đối tượng, có thể gọi bằng cách sử dụng tên lớp

Ứng dụng của Singleton Pattern trong thực tế: Đây có thể coi là một trong những pattern được sử dụng nhiều nhất trong các pattern, góp mặt trong hầu hết các thư viện lập trình. Singleton Pattern xuất hiện trong một số java core như java.lang.Runtime, java.lang.awt.Desktop (chỉ có một object được tạo).
Bấm vào đây để trở lại mục lục.

1.4 Builder Pattern

Buider Pattern là gì? Builder Patter là một pattern thuộc nhóm khởi tạo cho phép ta tách việc khởi tạo một đối phức tạp khỏi biểu diễn của chúng, qua đó ta có thể có quá trình khởi tạo tương tự từ những biểu diễn khác nhau.

Nói cách khác, Builder Pattern cho phép ta xây dựng một đối tượng phức tạp từng bước một.

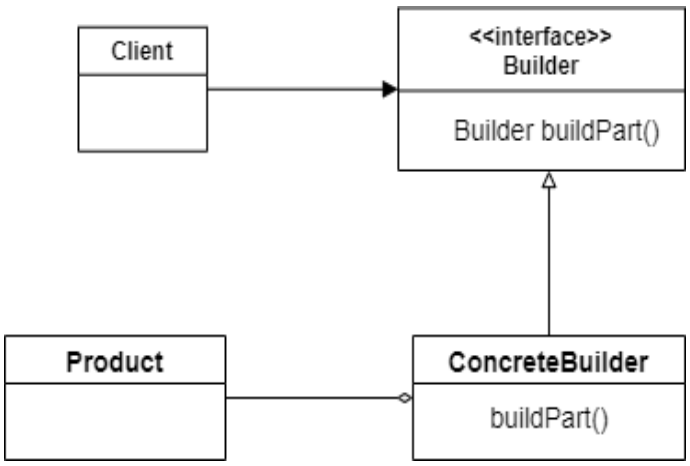
Mục đích sử dụng: Builder Pattern được sử dụng khi ta cần khởi tạo một đối tượng có nhiều thuộc tính mà không phải thuộc tính nào cũng cần bắt buộc có giá trị. Hoặc khi trong một đối tượng có quá nhiều hàm Constructor (hàm dựng), ta cũng nên dùng Builder Pattern. Ngoài ra, pattern này cũng có thể được sử dụng để tạo ra immutable object (đối tượng không thể thay đổi được).
Ví dụ: Khi ta muốn khởi tạo một đối tượng có nhiều thuộc tính:

- Nếu sử dụng constructor để khởi tạo, chúng ta sẽ phải viết thành một hàng rất dài với các đối số truyền vào khác kiểu, gây rối mắt và dễ nhầm giá trị.
- Chúng ta có thể nghĩ đến việc sử dụng nhiều Constructor (overload[7] các constructor). Tuy nhiên như vậy sẽ chỉ càng khiến ta gặp thêm rắc rối về thứ tự khởi tạo các giá trị cũng như dễ nhầm lẫn giữa những constructor có các tham số cùng kiểu dữ liệu.

Lúc này, việc sử dụng Builder Pattern sẽ giải quyết được vấn đề một cách hiệu quả và nhanh chóng, giúp code của chúng ta dễ đọc hơn.

Sơ đồ cấu trúc của Builder Pattern Sơ đồ cấu trúc của Builder Pattern có thể biểu diễn như sau:

[7] overload: nạp chồng, chỉ việc nhiều phương thức có cùng tên nhưng khác tham số cần truyền



Ta có thể giải thích sơ đồ này như sau:

- Product là đối tượng cần tạo, là một đối tượng phức tạp với nhiều trường thuộc tính.
- Builder được thiết kế là interface (hoặc lớp trừu tượng), có nhiệm vụ khai báo các phương thức dùng để tạo đối tượng theo từng phần. Ta thiết kế nó theo nguyên tắc "code to an interface", cho phép nó che giấu cách tạo ra từng phần cũng như để nó trở thành supertype của các lớp ConcreteBuider bên dưới.
- ConcreteBuilder: implements từ interface Builder, nó sẽ phải ghi đè lại các buildPart() để tạo ra các thành phần riêng biệt cho các đối tượng khác nhau.
- Khi muốn tạo ra đối tượng, Client sẽ gọi đến Builder để tạo các thành phần của đối tượng đó.

Code minh họa Builder Pattern: Chúng ta sẽ cùng minh họa bài toán khởi tạo đối tượng bằng Builder Pattern qua bài toán tạo ra chiếc điện thoại: Ta có lớp Phone gồm 6 thuộc tính cùng hàm constructor như sau:

```
    // Phone.java
public class Phone {
   private String os;
   private String name;
   private String brand;
   private double screenSize;
   private double battery;
   private double price;

   public Phone(String os, String name, String brand, double screenSize,
      double battery, double price) {
      this.os = os;
      this.name = name;
```

```
        this.brand = brand;
        this.screenSize = screenSize;
        this.battery = battery;
        this.price = price;
    }
    @Override
    public String toString() {
        return "Phone [os=" + os + ", name=" + name + ", brand=" + brand +
            ", screenSize=" + screenSize + ", battery="
              + battery + ", price=" + price + "]";
    }
}
```

Khi này muốn khởi tạo một chiếc iphone7, bên lớp main chúng là sẽ viết:

```
    // TestPhone.java
public class TestPhone {
    public static void main(String[] args) {
        Phone iPhone7 = new Phone("IOS", "Iphone 7", "Apple", 5.5, 1960,
            7000000);
    }
}
```

Như chúng ta thấy, dù mới chỉ có 6 thuộc tính thôi nhưng ta có thể thấy việc truyền đối số vào constructor đã rất rối mắt, chúng ta còn cần phải nhớ thứ tự các tham số được truyền vào để đảm bảo đối tượng được tạo ra đúng thông tin. Ngoài ra, có thể chúng ta không muốn khởi tạo giá trị cho tất cả các thuộc tính, nên việc sử dụng constructor gặp nhiều trở ngại.

Để khắc phục điều này, chúng ta có thể nghĩ đến việc sử dụng setter(). Tuy nhiên chúng ta cần nhớ rằng, iphone 7 ở đây là một đối tượng đã được tung ra thị trường, nên các thông số của nó sẽ không thay đổi trong suốt quá trình lưu hành (ngoại trừ giá cả). Nhưng với Builder Pattern, chúng ta có thể giải quyết được việc này:

- Chúng ta sẽ tạo ra một interface PhoneBuilder hiển thị tên của các hành động tạo ra thành phần của đối tượng Phone chúng ta cần tạo với kiểu trả về là PhoneBuilder.
- Riêng phương thức addPrice() cho phép ta truyền giá trị vào tham số "price" để có thể thay đổi giá cả theo thị trường.
- Phương thức build() có kiểu trả về là kiểu của đối tượng cần tạo, cụ thể ở đây là kiểu Phone. Phương thức build() này sẽ đóng vai trò hoàn thành việc xây dựng đối tượng.

:

```
    // PhoneBuilder.java
public interface PhoneBuilder {

   public PhoneBuilder addOs();

   public PhoneBuilder addName();

   public PhoneBuilder addBrand();

   public PhoneBuilder addScreensize();

   public PhoneBuilder addBattery();

   public PhoneBuilder addPrice(double price);

   public Phone build();
}
```

Tiếp theo, lớp Iphone7Builder implements từ interface PhoneBuilder. Ta sẽ tạo các thuộc tính giống hệt thuộc tính của lớp Phone trong lớp này, và với mỗi phương thức "add...()", ta sẽ gán giá trị cho các thuộc tính và trả về đối tượng hiện tại "return this". Ta làm được điều này vì các phương thức đó đều có cùng kiểu trả về là PhoneBuilder mà PhoneBuilder là supertype của Iphone7Builder (nhờ code to an interface).

Phương thức build() sẽ tạo ra một đối tượng kiểu Phone mới bằng cách tạo ra nó bằng constructor của lớp Phone sau đó "return". Tuy nhiên do các đối số được chúng ta truyền giá trị vào tham số của constructor thông qua tên biến nên sẽ không gây nhầm lẫn và rối mắt.

```
    // Iphone7Builder.java
public class Iphone7Builder implements PhoneBuilder {
   private String os;
   private String name;
   private String brand;
   private double screenSize;
   private double battery;
   private double price;

   public PhoneBuilder addOs() {
      this.os = "IOS";
      return this;
```

```
    }

    public PhoneBuilder addName() {
        this.name = "Iphone 7";
        return this;
    }

    public PhoneBuilder addBrand() {
        this.brand = "Apple";
        return this;
    }

    public PhoneBuilder addScreensize() {
        this.screenSize = 5.5;
        return this;
    }

    public PhoneBuilder addBattery() {
        this.battery = 1960;
        return this;
    }

    public PhoneBuilder addPrice(double price) {
        this.price = price;
        return this;
    }

    public Phone build() {
        return new Phone(os, name, brand, screenSize, battery, price);
    }
}
```

Lớp test: Ta có thể thấy rõ ràng việc khởi tạo đổi tượng bây giờ trở nên dễ nhìn hơn rất nhiều, ta có thể sử dụng" .add...()" liên tiếp là do các phương thức đó đều trả về cùng 1 đối tượng. Phương thức build() sẽ được gọi cuối cùng để hoàn tất quá trình khởi tạo này

---

```
    // TestPhone.java
public class TestPhone {
    public static void main(String[] args) {
        PhoneBuilder iphone7Builder = new Iphone7Builder();
        Phone iphone7New = iphone7Builder.addOs().addName().addBrand()
                                         .addScreensize().addBattery()
```

```
                                    .addPrice(7000000).build();
        Phone iphone7Old = iphone7Builder.addOs().addName().addBrand()
                                    .addScreensize().addBattery()
                                    .addPrice(3000000).build();
        System.out.println(iphone7New);
        System.out.println(iphone7Old);
    }
}
```

Chú ý: Việc thêm .build() ở cuối là cực kỳ quan trọng vì .build() mới tạo ra đối tượng kiểu Phone.
Output:

```
Phone [os=IOS, name=Iphone 7, brand=Apple, screenSize=5.5,
    battery=1960.0, price=7000000.0]
Phone [os=IOS, name=Iphone 7, brand=Apple, screenSize=5.5,
    battery=1960.0, price=3000000.0]
```

Ứng dụng của Buider Pattern trong thực tế: Buider Pattern được sử dụng trong một số thư viện Java như java.lang.StringBuilder ...

2 Structural Pattern

Là nhóm Design Pattern quan tâm đến việc thiết lập quan hệ giữa các lớp và đối tượng để có thể tạo ra một cấu trúc lớn hơn. Các Structural Pattern có thể kể đến như Decorator, Adapter, Bridge, FlyWeight, Composite...
Bấm vào đây để trở lại mục lục.

2.1 Adapter Pattern

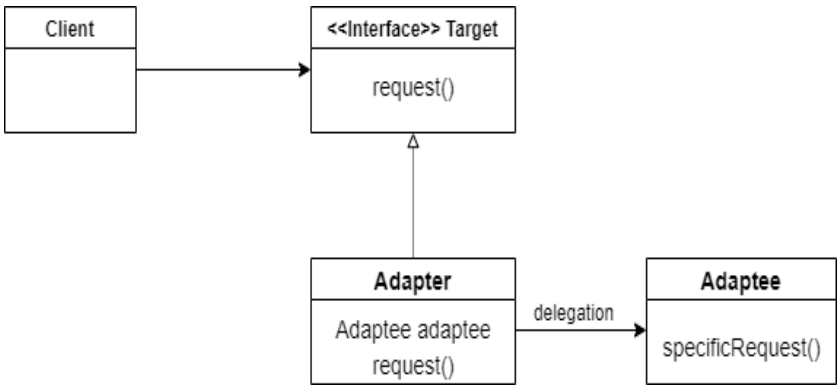
Adapter Pattern là gì? Adapter Pattern là một pattern cho phép chúng ta chuyển đổi interface của một class sang một interface khác mà chúng ta mong muốn. Pattern này cho phép các interface không liên quan đến nhau có thể hoạt động cùng nhau, thông qua một bộ chuyển đổi gọi là Adapter.

Mục đích của việc sử dụng Adapter Pattern Adapter Pattern giữ vai trò trung gian giữa các lớp, chuyển đổi interface của lớp này thành lớp khác sao cho chúng thích hợp với interface chúng ta mong muốn. Ta sử dụng Adapter khi không muốn thay đổi code của các lớp có sẵn mà vẫn muốn chúng có thể giao tiếp tốt với nhau. Adapter còn giúp ta có thể tái sử dụng các đoạn code.

Ví dụ minh họa về Adapter Pattern: Trong thực tế, Adapter Pattern rất thông dụng, đặc biệt là đối với các thiết bị điện tử

- Bộ chuyển đổi từ ổ cắm 3 lỗ sang 2 lỗ, giúp ta có thể dễ dàng sử dụng các thiết bị 2 chân cắm mà không cần mua ổ điện khác.
- Bộ chuyển nguồn ở máy tính giúp ta chuyển đổi từ dòng điện xoay chiều 220V sang dòng điện 1 chiều để máy tính có thể sử dụng được mà không cần can thiệp vào ổ điện hay máy tính.

Mô hình cấu trúc của Adapter Pattern Mô hình cấu trúc của Adapter Pattern



Ta có thể giải thích mô hình cấu trúc này như sau:

- Client đang muốn làm việc với Adaptee, tuy nhiên Client chỉ được thiết kế để có thể làm việc được với Target (không tương thích giao diện với Adaptee).
- Lớp Target sẽ được thiết kế là 1 interface (theo nguyên tắc code to an interface) sau đó các lớp Adapter sẽ implements lại Target, bằng cách này, Target sẽ là supertype của tất cả các Adapter, cho phép ta tạo ra nhiều Adapter tùy vào mục đích sử dụng.
- Adapter được thiết kế để chuyển giao diện từ Adaptee sang Target, bởi vậy Adapter phải vừa có giao diện của Target nhưng hình hài lại là của Adaptee, ta sẽ đạt được điều này bằng cách sử dụng kỹ thuật ủy quyền (delegate), các hành động của Adapter sẽ được ủy quyền sang cho Adaptee thực hiện.
- Bằng cách này, chúng ta có thể khiến client làm việc được với Adaptee mà không làm thay đổi các đoạn code đã có sẵn.

Để cho mô hình trở nên dễ hiểu hơn, chúng ta sẽ đến với phần code minh họa và giải thích.

Code minh họa và giải thích: Đầu tiên, ta có sẵn một mô hình các loài vịt gồm interface Duck với 2 hành vi là fly() và quack(), một lớp con MallardDuck được implements từ interface Duck

```
    // Duck.java
public interface Duck {
   public void quack();

   public void fly();
}
    // MallardDuck.java
public class MallardDuck implements Duck {
   public void quack() {
      System.out.println("Quack");
   }

   public void fly() {
      System.out.println("I'm flying");
   }
}
```

Tiếp sau đó, ta muốn thêm loài gà tây (Turkey) vào mô hình Duck, dù bản chất chúng không liên quan đến nhau.

```
    // Turkey.java
public interface Turkey {
   public void gobble();

   public void fly();
}
    // WildTurkey.java
public class WildTurkey implements Turkey {
   public void gobble() {
      System.out.println("Gobble gobble");
   }

   public void fly() {
      System.out.println("I'm flying a short distance");
   }
}
```

Để cho mô hình vịt vẫn có thể quản lý tốt được loài gà tây, lúc này ta sẽ cần một lớp Adapter làm cho loài gà tây trở nên giống loài vịt, ta sẽ thực hiện điều đó dựa theo mô

hình thiết kế Adapter Pattern bên trên:

- Ta sẽ tạo lớp TurkeyAdapter (Adapter) implements từ interface Duck (Target).
- Sau đó, bằng việc sử dụng kỹ thuật Delegate, ta sẽ thực hiện các hành vi đã được implements từ Duck trong TurkeyAdapter thông qua Turkey (Adaptee). Ta làm điều này bằng cách tạo một Turkey turkey trong lớp TurkeyAdapter, sau đó với các phương thức như quack() hay fly(), ta sẽ dùng turkey.gobble() hay turkey.fly(), như vậy ta đã hoàn toàn liên kết được giao diện của hai lớp Duck và Turkey.
- Tức là các hành vi của TurkeyAdapter bề ngoài là của Duck nhưng thực chất các hành động đó được thực hiện theo cách của Turkey.

Điều đó được thực hiện trên code một cách cụ thể như sau:

```
    // TurkeyAdapter.java
public class TurkeyAdapter implements Duck {
   Turkey turkey;

   public TurkeyAdapter(Turkey turkey) {
      this.turkey = turkey;
   }

   public void quack() {
      turkey.gobble();
   }

   public void fly() {
      for (int i = 0; i < 2; i++) {
         turkey.fly();
      }
   }
}
```

Như vậy, khi gọi đến fly() của TurkeyAdapter, ta sẽ có thể thực hiện được hành vi fly() của một Turkey chứ không phải của một Duck, tương tự với hành vi quack(). Ta sẽ xem qua lớp DuckTestDrive.java để kiểm chứng điều này.

```
    // DuckTestDrive.java
public class DuckTestDrive {
   public static void main(String[] args) {
      Duck duck = new MallardDuck();
      Turkey turkey = new WildTurkey();
      Duck turkeyAdapter = new TurkeyAdapter(turkey);
```

```
        System.out.println("The Turkey says...");
        turkey.gobble();
        turkey.fly();

        System.out.println("\nThe Duck says...");
        testDuck(duck);

        System.out.println("\nThe TurkeyAdapter says...");
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

Output:

```
The Turkey says...
Gobble gobble
I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance
I'm flying a short distance
```

Như vậy, ta đã có thể quản lý Turkey chung với Duck. Tuy nhiên cũng cần lưu ý, có một số trường hợp trong lớp Adapter có những phương thức khác với trong lớp Target, khi đó chúng ta sẽ không thể dùng supertype Target để gọi tới các phương thức đó được. Thay vào đó, chúng ta cần tiến hành Downcast[8] (ép kiểu xuống), chuyển kiểu Target xuống Adapter.

Ví dụ thực tế sử dụng Adapter Pattern: Trong thực tế, Adapter Pattern cũng được sử

[8]downcast: ép kiểu xuống, là việc ta chuyển kiểu dữ liệu của một đối tượng từ supertype thành subtype, chỉ thực hiện được khi bản chất 2 đối tượng như nhau

dụng khá nhiều, đặc biệt là trong các dự án khi ta cần kết nối chương trình hiện tại với một API/thư viện đã có sẵn bên ngoài. Một dự án thực tế thường rất lớn nên đôi khi việc sử dụng những thứ đã có trước là không thể tránh khỏi, sử dụng Adapter Pattern sẽ giúp ta tránh được việc phải sửa lại toàn bộ code ở các lớp có sẵn.
Bấm vào đây để trở lại mục lục.

2.2 Bridge Pattern

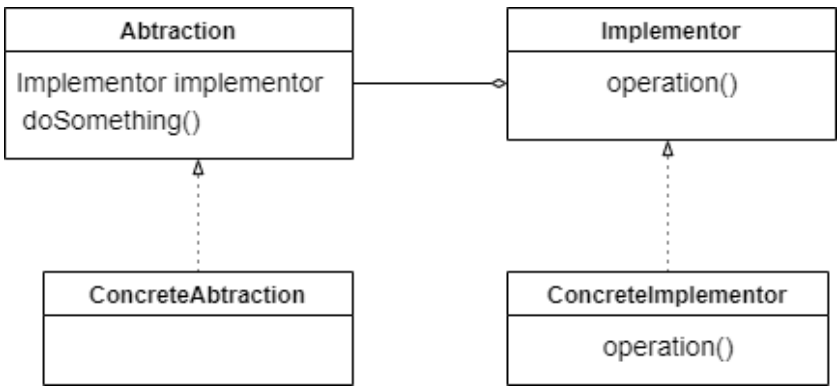
Bridge Pattern là gì? Bridge Pattern là một pattern thuộc nhóm cấu trúc, nó cho phép tách một phần trừu tượng (abstraction) ra khỏi phần triển khai (implementation) của nó, qua đó 2 phần này có thể độc lập với nhau.

Ta có thể đạt được điều này thông qua việc sử dụng kỹ thuật ủy quyền. Các phương thức hoạt động cần ghi đè ở lớp trừu tượng sẽ được chuyển sang một lớp khác, lớp trừu tượng sẽ giữ một đối tượng của lớp thực hiện để có thể thực hiện các hành vi đó (compositon và delegation).

Mục đích của việc sử dụng Bridge Pattern: Sử dụng Bridge Pattern khi chúng ta muốn có thể thay đổi lớp trừu tượng (abtraction) và lớp thực hiện (implementation) một cách độc lập, không chịu quá nhiều ràng buộc. Hoặc trong một số trường hợp khác như:

- Khi lớp Abstraction và lớp Implementation đều cần được mở rộng bằng các sub-class. Lúc này việc tách ra sẽ rất hữu ích.
- Việc này cũng có thể giúp giảm độ phức tạp của code.
- Vẫn có thể che giấu được cách thực hiện cụ thể các lớp thực hiện do Client sẽ làm việc với lớp abstraction.

Sơ đồ cấu trúc của Bridge Pattern Sơ đồ của pattern này nhìn gần giống tên gọi của nó (cây cầu).



Trong đó:

- Abtraction và Implementor lúc này sẽ đều được thiết kế là các lớp trừu tượng (code to interface). Lớp Abtraction sẽ là supertype của các ConcreteAbstraction, các lớp con này sẽ sử dụng một tham chiếu có supertype là Implementor để thực hiện các phương thức được định ra.
- phương thức doSomething() sẽ được các ConcreteAbstraction ghi đè lại và thực hiện theo cách implementor.operation();
- Implementor sẽ định ra các tác vụ cần thực hiện của Abtraction, sau đó các ConcreteImplementor sẽ implements lại nó để thực hiện các nhiệm vụ này.

Code minh họa: Ta sẽ minh họa việc sử dụng Bridge Pattern trong thiết kế một chương trình liên quan đến Game và Account. Chương trình ban đầu gồm những thành phần như sau:

- Ta muốn thiết kế lớp Game có phương thức mở tài khoản openAccount() nhằm tạo loại tài khoản chúng ta mong muốn.
- Hiện tại mới chỉ có 2 tựa game đó là HornorGame (game kinh dị) và FunnyGame (game vui). Tài khoản cũng được chia làm 2 loại là UserAccount (tài khoản người dùng) và ClientAccount (tài khoản khách).
- Trong thực tế, 2 loại tài khoản này có chức năng rất khác nhau tuy nhiên trong ví dụ dưới đây, ta sẽ thiết kế một cách đơn giản nhằm thể hiện cấu trúc của Bridge Pattern.

Nếu theo cách thiết kế thông thường, ta sẽ thiết kế 2 lớp HornorGame và FunnyGame được kế thừa từ lớp Game, sau đó với mỗi lớp FunnyGame và HornorGame, 2 lớp ClientAccount và UserAccount sẽ kế thừa chúng. Tuy nhiên có thể thấy cách làm này rất hạn chế.

Nó làm tăng số class ta cần phải tạo lên và mỗi khi mở thêm một loại game mới (ví dụ như ActionGame), ta lại phải thiết kế thêm UserAccount và ClientAccount kế thừa mỗi lớp đó, khiến cho chương trình trở nên khó mở rộng. Với cách thiết kế theo Bridge Pattern, ta sẽ cấu trúc lại chương trình như sau:
Thay vì để các loại tài khoản kế thừa từ các loại game cụ thể, ta sẽ tạo ra một lớp Account trừu tượng riêng và các loại tài khoản được kế thừa từ nó (code to an interface). Lớp Account sẽ khai báo một phương thức trừu tượng là createAccount();

```
    // Account.java
abstract public class Account {
   abstract void createAccount();
}
    // ClientAccount.java
```

```
public class ClientAccount extends Account {
   public void createAccount() {
      System.out.println("Create a client account");
   }
}
    // UserAccount.java
public class UserAccount extends Account {
   public void createAccount() {
      System.out.println("Create an user account");
   }
}
```

Các lớp UserAccount và ClientAccount sẽ ghi đè lại createAccount() từ lớp Account để thực hiện phương thức theo những cách khác nhau. Tiếp theo là lớp trừu tượng Game. Ta sẽ để một tham chiếu tới Account trong lớp Game. Các lớp con của Game sẽ ủy quyền cho tham chiếu đó để thực hiện phương thức openAccount().

```
    // Game.java
public abstract class Game {
   protected Account account;

   public Game(Account account) {
      this.account = account;
   }

   public abstract void openAccount();
}

    // HornorGame.java
public class HornorGame extends Game {
   public HornorGame(Account account) {
      super(account);
   }

   public void openAccount() {
      System.out.println("Open an account of Hornor game");
      account.createAccount();
   }
}

    // FunnyGame.java
public class FunnyGame extends Game {
```

```
    public FunnyGame(Account account) {
        super(account);
    }

    public void openAccount() {
        System.out.println("Open an account of Funny game");
        account.createAccount();
    }
}
```

Hàm constructor của các lớp game sẽ cho phép truyền vào một biến kiểu Account, điều này cho phép tạo ra tài khoản game tương ứng tùy theo đối số truyền vào.

Lớp test:

```
    // TestGame.java
package bridge;

public class TestGame {
    public static void main(String[] args) {
        Account client = new ClientAccount();
        Account user = new UserAccount();

        Game hornorGame = new HornorGame(client);
        hornorGame.openAccount();
        Game funnyGame = new FunnyGame(user);
        funnyGame.openAccount();
    }
}
```

Output:

```
Open an account of Hornor game
Create a client account
Open an account of Funny game
Create an user account
```

Ví dụ Bridge Pattern được sử dụng trong thực tế: Bridge Pattern đặc biệt hữu dụng trong các ứng dụng đa nền tảng (cross platform app), trong các ứng dụng Social media...
Bấm vào đây để trở lại mục lục.

2.3 Decorator Pattern

Decorator Pattern là gì? Decorator Pattern (trang trí) là một pattern thuộc nhóm cấu trúc cho phép ta gắn thêm các trách nhiệm bổ sung cho đối tượng hiện tại một cách linh hoạt. Hay nói cách khác, ta có thể thêm chức năng mới vào đối tượng mà tránh làm thay đổi các lớp khác (do thừa kế.

Decorator pattern hoạt động giống như một vỏ bọc, nó sẽ " bọc" các đối tượng bổ sung xung quanh đối tượng cũ, mỗi khi có thêm tính năng mới, các đối tượng hiện tại sẽ được" bọc" trong một đối tượng mới, lớp chứa đối tượng mới này có thể gọi là Decorator class.

Mục đích sử dụng Decorator pattern: Pattern này được sử dụng khi ta muốn mở rộng đối tượng (thêm các tính năng mới...) mà không làm thay đổi chúng. Ngoài ra, pattern này còn cho phép ta thêm hoặc bớt tính năng của một đối tượng tại run-time (điều mà kế thừa không thực hiện được). Đây là một pattern mạnh trong việc thiết kế đối tượng sao cho có thể mở rộng linh hoạt.

Ví dụ minh họa về Decorator Pattern: Chúng ta có thể hiểu đơn giản về Decorator Pattern qua ví dụ về trà sữa. Thông thường khi mua trà sữa, chúng ta sẽ chọn thêm các topping như trân châu, pudding, kem, matcha...

- Ta có thể hiểu trà sữa giống như đối tượng của chúng ta, và các topping giống như các decorator class. Ban đầu chúng ta chỉ có đối tượng trà sữa, khi đổ thêm trân châu vào, nó sẽ thành trà sữa trân châu, đổ thêm đường đen, nó sẽ thành trà sữa trân châu đường đen.
- Và chúng ta hoàn toàn có thể thay đổi các topping được thêm vào bằng cách lựa chọn có thêm topping đó hay không. Ngoài ra việc tính topping riêng cũng sẽ khiến người bán dễ dàng hơn khi tính tiền, vì khi giá của các topping thay đổi, ta chỉ cần sửa đổi giá của topping thôi.

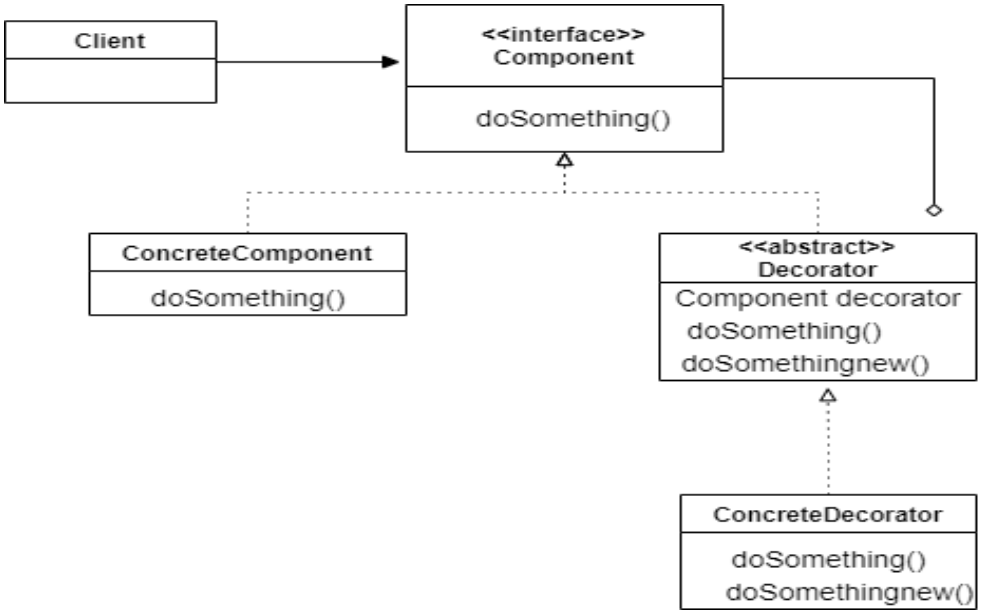
Nếu như ngay từ đầu, ta tạo ra các loại trà sữa kèm topping quá cụ thể như trà sữa kem cheese trân châu trắng...và ấn định giá của chúng, mỗi khi các topping thay đổi giá tiền, ta sẽ phải thay đổi giá của từng loại trà sữa đó, điều này thật sự rất mất thời gian.

Sơ đồ cấu trúc của Decorator Pattern Sơ đồ cấu trúc của Decortor Pattern được thể hiện như sau:

Trong đó:

- Component và Decorator sẽ được ta thiết kế theo nguyên tắc "code to an interface" nhằm tạo ra các kiểu supertype thuận tiện cho việc sử dụng. Nó cùng giúp che giấu cách thực hiện hành vi doSomething() với Client.

- Decorator có chứa một tham chiếu đến Component nên ta sẽ thiết kế Decorator là abstract class, và nó được implements từ interface Component nên nó cũng phải có phương thức doSomething(). Cùng với đó, nó sẽ có thể có thêm các phương thức mới (thường là abstract method), các phương thức này sẽ được các lớp con của nó thực hiện.



- Các lớp con ConcreteComponent của Component sẽ thực hiện ràng buộc về hành vi doSomething() của interface Component.
- Các lớp ConcreteDecorator sẽ thực hiện các hành vi ràng buộc có trong lớp Decorator.

Code minh họa về Decorator Pattern: Trong phần code minh họa này, chúng ta sẽ thử giải quyết bài toán thiết kế chương trình thanh toán trà sữa bên trên. Giả sử của hàng của chúng ta lúc này mới mới đang chỉ có 2 loại trà sữa đó là MangoMilktea (trà sữa xoài) và Blacktea (hồng trà sữa), với một số loại topping như BlackBubble (trân châu đen), OreoCream (kem Oreo), Sugar (đường).

- Ta sẽ thiết kế một lớp trừu tượng Milktea là supertype của MangoMilktea và Blacktea, trong lớp Milktea sẽ khai báo một phương thức trừu tượng là cost() dùng để tính tiền. Milktea sẽ có một thuộc tính là String description để mô tả cốc trà, cùng với đó là phương thức getDescription().
- Các lớp con MangoMilktea và Blacktea phải ghi đè lại phương thức cost() theo ràng buộc. Trong constructor của các lớp con, ta sẽ gán lại giá trị cho thuộc tính description để mô tả cốc trà cụ thể hơn.

```
    // Milktea.java
abstract public class Milktea {
   protected String description = "Unknowm Milktea";

   public String getDescription() {
      return description;
   }

   abstract public double cost();
}

    // MangoMilktea.java
public class MangoMilktea extends Milktea {
   public MangoMilktea() {
      description = "Mango Milktea";
   }

   public double cost() {
      return 8000;
   }
}

    // Blacktea.java
public class Blacktea extends Milktea {
   public Blacktea() {
      description = "Black tea";
   }

   public double cost() {
      return 15000;
   }
}
```

Ứng với sơ đồ, các lớp Milktea, MangoMilktea và Blacktea chính là Component và các ConcreteComponent. Tiếp theo về phần topping, chúng ta sẽ thiết kế một lớp trừu tượng là ToppingDecorator là superclass của tất cả các topping, cho phép ta trỏ đến bất kỳ topping nào. Bên trong lớp ToppingDecorator sẽ chứa một tham chiếu có kiểu Milktea (compositon).

```
    // ToppingDecorator.java
abstract public class ToppingDecorator extends Milktea {
```

```
    protected Milktea milktea;

    public abstract String getDescription();

    public abstract double cost();
}
```

Tiếp theo là các lớp topping cụ thể, chúng sẽ ghi đè lại getDescription() và cost() để thêm mô tả về cốc trà sữa cùng như cộng thêm giá mỗi khi thêm topping. Ta có thể dễ dàng đạt được điều này nhờ tham chiếu "milktea" đã được chứa trong ToppingDecorator. Đây chính là sức mạnh của việc sử dụng composition và delegation thay vì thừa kế.

Trong constructor của lớp này, ta sẽ truyền vào một đối số kiểu Milktea và gán tham chiếu "milktea" hiện tại bằng giá trị đối số truyền vào. Như vậy ta có thể xác định được lại trà được thêm topping là loại gì (MangoMilktea hay Blacktea ...).

```
    // BlackBubble.java
public class BlackBubble extends ToppingDecorator {
    public BlackBubble(Milktea milktea) {
        this.milktea = milktea;
    }

    public String getDescription() {
        return milktea.getDescription() + ", BlackBubble";
    }

    public double cost() {
        return milktea.cost() + 10000;
    }
}

    // Sugar.java
public class Sugar extends ToppingDecorator {
    public Sugar(Milktea milktea) {
        this.milktea = milktea;
    }

    public String getDescription() {
        return milktea.getDescription() + ", Sugar";
    }

    public double cost() {
        return milktea.cost() + 2000;
```

```
    }
}

    // OreoCream
public class OreoCream extends ToppingDecorator {
    public OreoCream(Milktea milktea) {
        this.milktea = milktea;
    }

    public String getDescription() {
        return milktea.getDescription() + ", OreoCream";
    }

    public double cost() {
        return milktea.cost() + 15000;
    }
}
```

Như vậy, mỗi khi cần thêm topping nào vào trà sữa, ta chỉ cần thêm các ToppingDecorator vào Milktea là được. Khi các topping thay đổi về giá, ta chỉ cần sửa lại phần cost() ở các lớp con. Ta cũng có thể dễ dàng mở rộng chương trình.
Lớp Test: Ta sẽ sử dụng các Decorator để "bọc" thêm cho đối tượng cần thêm như sau:

```
    // TestMilktea.java
public class TestMilktea {
    public static void main(String[] args) {
        Milktea milktea1 = new Blacktea();
        System.out.println(milktea1.getDescription()
                + String.format(" %.2f", milktea1.cost()));

        // Use decorator pattern
        milktea1 = new OreoCream(new Sugar(milktea1));
        System.out.println(milktea1.getDescription()
                + String.format(" %.2f", milktea1.cost()));
    }
}
```

Vì supertype lớn nhất của các topping đều là Milktea, với cấu trúc của hàm constructor của chúng, chúng ta có thể liên tục truyền thêm các topping vào bên trong để thêm các tính năng cho đối tượng.
Output:

```
Black tea 15000.00
Black tea, Sugar, OreoCream 32000.00
```

Ứng dụng của Decorator Pattern trong thực tế: Một trong những ứng dụng đáng nói của Decorator Pattern là hầu như toàn bộ API java.io được xây dựng dựa theo pattern này. Với abstract component của java.io là InputStream và Decorator là FilterInputStream.

3 Behavioral Pattern

Là nhóm Design Pattern dùng trong việc thực hiện hành vi của đối tượng và sự giao tiếp của các đối tượng với nhau. Các pattern trong nhóm này có thể kể đến như: Strategy, Observer, Command, Iterator...
Bấm vào đây để trở lại mục lục.

3.1 Strategy Pattern

Strategy Pattern là gì? Strategy Pattern (Chiến thuật) là một pattern cho phép định nghĩa tập hợp các chiến thuật (thuật toán...), đóng gói chúng lại, chúng ta có thể thay đổi các chiến thuật (thuật toán...) một cách đơn giản trong đối tượng. Strategy pattern cũng cho phép các chiến thuật biến đổi độc lập khi người dùng sử dụng chúng.

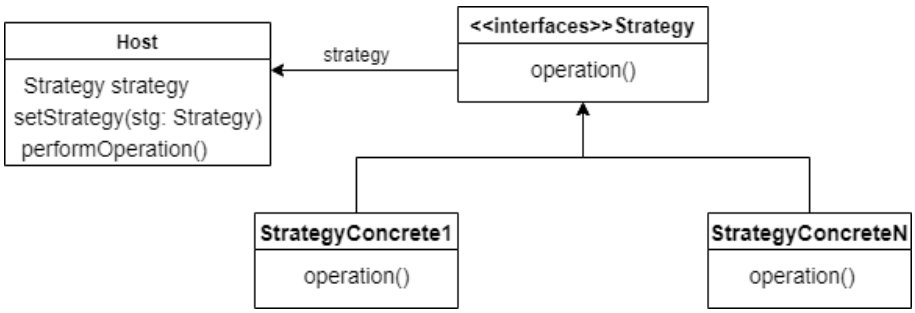
Mục đích sử dụng: Strategy Pattern giúp tách rời phần xử lý một chức năng cụ thể ra khỏi đối tượng, sau đó tạo ra một tập hợp các chiến thuật để xử lý chức năng đó, tùy vào từng đối tượng, ta sẽ chọn ra chiến thuật phù hợp. Mẫu thiết kế này thường được sử dụng để thay thế cho tính kế thừa khi ta muốn chấm dứt việc theo dõi và chỉnh sửa qua một subclass(lớp con).
Ngoài ra Strategy Pattern còn giúp chúng ta có thể dễ dàng thay đổi và mở rộng các chiến thuật cụ thể mà không ảnh hưởng đến toàn bộ chương trình cũng như che giấu sự phức tạp của các thuật toán bên trong.

Ví dụ minh họa về Strategy Pattern Chúng ta có thể lấy ví dụ đơn giản về Strategy Pattern trong thực tế thông qua cách bay của loài chim. Có loài chim bay cao, có loài chim bay thấp, bay là là mặt đất, bay trung bình.

- Ta có thể coi mỗi khả năng bay đó là chiến thuật, và các chiến thuật đó có thể gói gọn lại thành hành vi bay của các loài chim, bay cao, bay thấp đều có thể gọi là hành vi bay, và khi nhắc đến hành vi bay, ta sẽ liệt kê được các hành vi bay cao, bay thấp...

- Mỗi loài chim khác nhau sẽ có thể sử dụng một chiến thuật khác nhau, và chúng biểu hiện chiến thuật đó thông qua hành vi bay của mình.
- Ví dụ như loài chim hoang dã có khả năng bay cao, ta có thể coi nó đang sử dụng chiến thuật bay cao, khác với loài chim trời sử dụng chiến thuật bay thấp.

Cấu trúc của Strategy Pattern: Cấu trúc của Strategy Pattern được biểu diễn thông qua mô hình cấu trúc đơn giản sau:



- Trong đó: Host sẽ nhận yêu cầu từ Client, sau khi nhận được yêu cầu, Host sẽ ủy quyền (delegate) cho Strategy để thực hiện các chiến thuật tương ứng.
- Interface Strategy: liệt kê các chiến thuật có thể được sử dụng, việc các chiến thuật đó được thực hiện như nào được triển khai cụ thể trong các lớp StrategyConcreteN được implements từ Strategy.
- StrategyConcreteN: Các chiến thuật (thuật toán) có thể được thực hiện. Để host có thể thực hiện được nhiều chiến thuật khác nhau mà không phải cụ thể một thuật toán nào, ta cần thiết kế một kiểu dữ liệu là supertype cho tất cả các thuật toán, đó là lý do ta thiết kế interface Strategy (code to an interface).
- Khi performOperation() được gọi tới: strategy.operation();
- Hàm setStrategy(Strategy stg) được sử dụng để có thể thay đổi chiến thuật khi run-time, chúng ta có thể triển khai chúng theo dạng .setStrategy(new StrategyConcreteN());

  Ví dụ: Khi ta đang muốn thực hiện khả năng bay của các loài chim như bay cao (FlyHigh), bay thấp (FlyLow), ta sẽ implements chúng từ interface FlyBehavior (hành vi bay), mỗi một khả năng bay(cao, thấp ...) ta sẽ coi là một chiến thuật, tùy từng loài chim sẽ sử dụng những chiến thuật khác nhau sao cho phù hợp. Bằng việc sử dụng hàm set, ta có thể thay đổi chiến thuật một cách dễ dàng.

Code minh họa Strategy Pattern Phần code minh họa, sơ đồ thiết kế và giải thích code trong một bài toán cụ thể dùng Strategy Pattern đã được trình bày thông qua ví dụ về vườn chim ở mục 4 phần I.
Chuyển đến phần code minh họa

Strategy Pattern được sử dụng trong thực tế: Trong thực tế, ta có thể tìm thấy Strategy Pattern ở rất nhiều chương trình, điển hình là thư viện List trong java. Ta có thể coi các phương thức như add, remove, set... là các chiến thuật.
Bấm vào đây để trở lại mục lục.

3.2 Observer Pattern

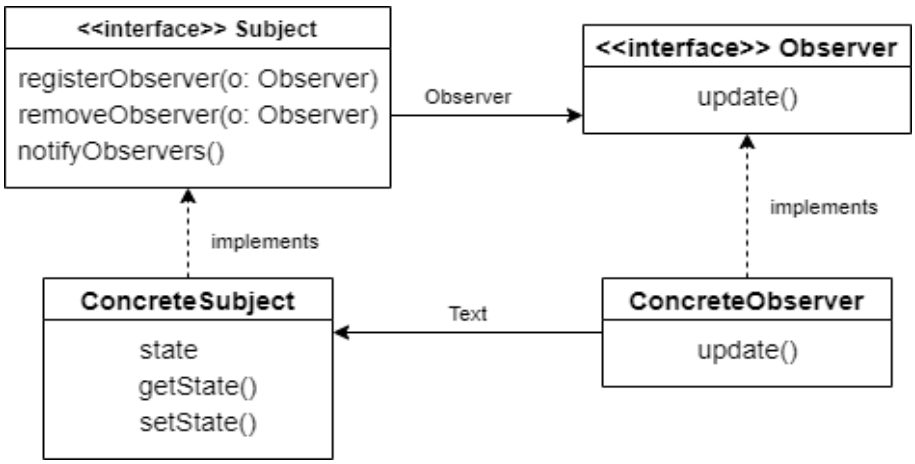
Observer Pattern là gì? Observer Pattern (Quan sát) là một pattern thuộc nhóm hành vi, nó định nghĩa mối phụ thuộc one - many (một nhiều) giữa các đối tượng với nhau, để khi một đối tượng thay đổi, tất cả các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động.

Mục đích của việc sử dụng Observer Pattern: Observer Pattern thường được sử dụng khi ta muốn sự thay đổi của một đối tượng được thông báo cho các đối tượng khác và sự thay đổi được cập nhật một cách tự động. Observer Pattern được sử dụng rất rộng rãi và phổ biến. Các Observer có thể đăng ký với hệ thống để nhận thông báo, và khi không cần nữa, chúng ta có thể gỡ chúng ra.
Có thể có nhiều Observer quan sát các subject.

Ví dụ minh họa về Observer Pattern: Chúng ta sẽ lấy ví dụ về Observer Pattern trong thực tiễn về mối quan hệ giữa cấp trên và cấp dưới trong một công ty. Trong một công ty đơn giản gồm 3 vị trí: Giám đốc, Quản lý, Nhân viên.

- Ta có thể coi quản lý là một Observer. Quản lý muốn biết tiến độ làm việc của nhân viên nên có thể yêu cầu nhân viên cập nhật tiến độ công việc cho mình (việc này giống như Observer đăng ký với hệ thống), mỗi khi tiến độ công việc của nhân viên thay đổi, thông báo sẽ được gửi cho quản lý, và quản lý sẽ báo cáo lên giám đốc (thực hiện hay đổi).
- Khi nhân viên (subject) đó rời công ty hoặc rời phòng ban, quản lý có thể hủy quan sát nhân viên đó.

Mô hình cấu trúc của Observer Pattern Để dễ hiểu hơn, chúng ta sẽ cùng xem qua sơ đồ cấu trúc của Observer Pattern:



Ta có thể giải thích sơ đồ này như sau: Đầu tiên, các subject sẽ được code to an interface thành một supertype Subject, trên interface này ít nhất phải khai báo 3 tác vụ:

- registerObserver(Observer o): dùng để các Observer có thể đăng ký quan sát Subject
- removeObserver(Observer o): dùng để các Observer có thể hủy đăng ký quan sát
- notifyObservers(): để gửi thông báo đến cho các Observer đang đăng ký quan sát subject khi Subject có sự thay đổi
- Các ConcreteSubject được implements từ Subject để thực hiện các ràng buộc có trên interface và triển khai một đối tượng cụ thể.

Observer cũng được code theo interface với một API là update() để khi nhận được thông báo từ Subject, nó có thể cập nhật gì đó. Các ConcreteObserver được implements từ Observer sẽ thực hiện các ràng buộc phương thức trên Observer.

Code minh họa và giải thích: Chúng ta sẽ thực hiện một chương trình đơn giản để mô tả cách triển khai và sử dụng của Observer Pattern.

Đầu tiên ta sẽ tạo interface Subject và interface Observer, trong Subject chứa 3 phương thức như đã trình bày ở phần sơ đồ, Observer chứa phương thức update()

```
    // Subject.java
public interface Subject {

   void registerObserver(Observer o);

   void removeObserver(Observer o);

   void notifyObservers();
}
    // Observer.java
public interface Observer {
```

```
    public void update(int value);
}
```

Tiếp theo, ta sẽ tạo lớp SimpleSubject implements từ Subject để thực thi các ràng buộc có trong Subject. Trong lớp con này có một List<Observer> để chứa các Observer đã đăng ký quan sát nó, vì Observer được tạo theo nguyên tắc Code to an interface nên ta có thể dễ dàng thực hiện điều này.

- Mỗi khi có thêm một Observer mới đăng ký quan sát, ta sẽ add thêm Observer đó vào List và ngược lại, khi một Observer bỏ quan sát, ta sẽ remove chúng khỏi List.
- Khi muốn thông báo đến các Observer đã đăng ký, ta chỉ cần duyệt các phần tử trong List và thực thi việc gửi thông báo.
- Ở phương thức notifyObserver(), ta đã thực hiện kỹ thuật delegate (ủy quyền) để cập nhật thông tin đến cho Observer bằng cách dùng một đối tượng có kiểu dữ liệu Observer và thực thi update().

Như vậy, chúng ta có thể add/remove các observer bất cứ lúc nào mà không cần phải thay đổi đối tượng khi thêm một loại mới. Ta sẽ cài đặt thêm một trường value và hàm setValue() để thay đổi giá trị cho một SimpleSubject.

```
    // SimpleSubject.java
import java.util.*;

public class SimpleSubject implements Subject {

    private List<Observer> observers;
    private int value = 0;

    public SimpleSubject() {
        observers = new ArrayList<Observer>();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    public void notifyObservers() {
        for (Observer observer : observers) {
```

```
            observer.update(value);
        }
    }

    public void setValue(int value) {
        this.value = value;
        notifyObservers();
    }
}
```

Lớp SimpleObserver sẽ được ta implements lại từ Observer để thực thi phương thức update() có trong Observer. Trong lớp SimpleObserver sẽ có:

- 2 trường gồm Subject simpleSubject (delegation) và int value.
- Hàm display() để in ra màn hình giá trị của biến value trên SimpleObserver hiện tại.
- Hàm dựng (constructor) cho phép truyền vào một Subject mà Observer muốn đăng ký quan sát, nhờ vào việc sử dụng delegate (ủy quyền), ta có thể đăng ký quan sát một Subject ngay trong hàm dựng này bằng cách sử dụng .registerObserver();
- Hàm update() sẽ cập nhật giá trị của biến value trong Observer mỗi khi giá trị value của Subject thay đổi (ta đã ủy quyền ở trên lớp SimpleSubject).

```
    // SimpleObserver.java
public class SimpleObserver implements Observer {

    private int value;
    private Subject simpleSubject;

    public SimpleObserver(Subject simpleSubject) {
        this.simpleSubject = simpleSubject;
        simpleSubject.registerObserver(this);
    }

    public void update(int value) {
        this.value = value;
        display();
    }

    public void display() {
        System.out.println("Value: " + value);
    }
}
```

Lớp test:

```
    // Example.java
public class Example {

  public static void main(String[] args) {
    SimpleSubject simpleSubject = new SimpleSubject();

    SimpleObserver simpleObserver = new SimpleObserver(simpleSubject);

    simpleSubject.setValue(80);

    simpleSubject.removeObserver(simpleObserver);

    simple.Subject.setValue(90);
  }
}
```

Output:

```
Value: 80
```

Như ta thấy, khi SimpleObserver vẫn đăng ký quan sát simpleSubject, việc sử dụng simpleSubject.setValue(80) sẽ gửi thông báo đến observer, observer sẽ cập nhật và thực thi hàm display() in ra kết quả Value: 80 ra màn hình. Sau khi hủy đăng ký, observer sẽ không còn quan sát simpleSubject nữa nên việc setValue(90) không được in ra output.

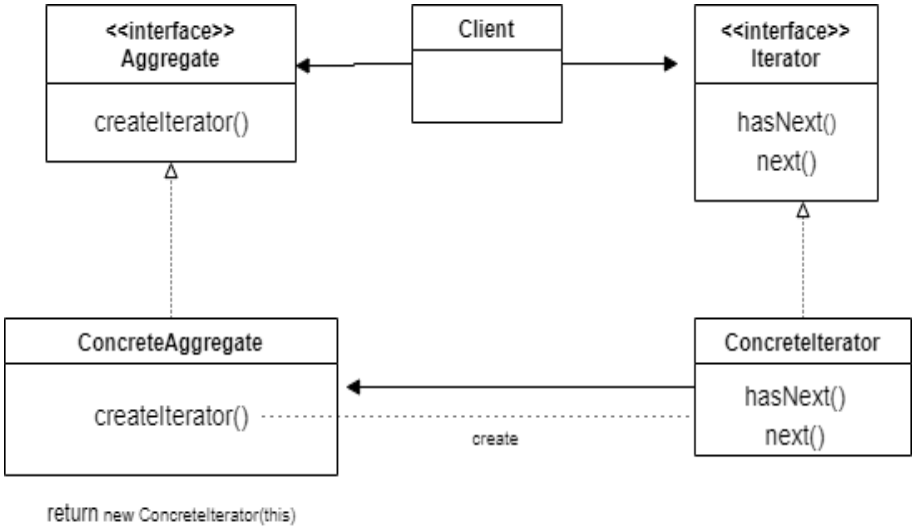
Ứng dụng trong thực tế của Obserber Pattern: Observer Pattern là một trong những pattern được sử dụng nhiều và phổ biến nhất hiện nay, hầu như trong hệ thống lớn nào cũng có sự xuất hiện của Observer Pattern. Ngay trong thư viện java.util, ta cũng có thể tìm thấy rất nhiều interface có thể kể đến như java.util.Observerble, java.util.Obserber... Bấm vào đây để trở lại mục lục.

3.3 Iterator Pattern

Iterator Pattern là gì? Iterator Pattern là một pattern thuộc nhóm hành vi cung cấp một cách thức truy cập tuần tự đến các phần tử của một đối tượng tổng hợp mà không làm lộ ra biểu diễn bên trong của nó.

Mục đích sử dụng Iterator Pattern: Pattern này được sử dụng khi ta muốn truy cập vào các phần tử các loại tập hợp khác nhau (array, arraylist...) với cùng một phương pháp, mà không cần biết cách mọi thứ được thể hiện bên trong như nào.

Sơ đồ cấu trúc Iterator pattern: Sơ đồ cấu trúc Iterator pattern có thể được biểu diễn một cách đơn giản như sau:



Trong đó:

- Các lớp Aggregate và Iterator đều được thiết kế sử dụng nguyên tắc "code to an interface", chúng có thể che giấu cách thực hiện hành vi với Client cũng như tạo ra các kiểu supertype.
- interface Iterator sẽ khai báo 2 phương thức hasNext() và next(). Phương thức hasNext() sẽ có kiểu là boolean, cho phép ta kiểm tra xem sau phần tử hiện tại có còn phần tử nào hay không, có thì trả về "true", không thì trả về "false". Phương thức next() sẽ trả lại phần từ tiếp theo, nếu như hasNext() là "true".
- các lớp ConcreteIterator sẽ implements Iterator và ghi đè hasNext() và next().
- Aggregate là một interface định nghĩa phương thức để tạo ra các đối tượng có kiểu Iterator. Phương thức này sẽ được các lớp con ConcreteAggregate ghi đè và thực hiện bằng cách trả lại ConcreteIterator tương ứng.

Chúng ta sẽ thấy rõ hơn cách mà Iterator truy cập được vào các phần tử của các tập hợp khác nhau qua phần code minh họa phía dưới.

Code minh họa Iterator Pattern: Chúng ta sẽ minh họa cách Iterator Pattern hoạt động thông qua bài toán gộp 2 menu sử dụng loại tập hợp khác nhau. Menu ăn sáng sử dụng mảng (array) để chứa các menu item trong khi menu ăn tối lại sử dụng ArrayList để chứa các menu item.
Aggregate trong bài toán này sẽ là interface Menu, các ConcreteAggrete sẽ là BreakfastMenu và DinnerMenu, tương ứng các ConcreteIterator là BreakfastMenuIterator và DinnerMenuIterator.

Đầu tiên, ta sẽ đến với lớp MenuItem, lớp này gồm những thuộc tính cần có của các dòng trong menu. Lớp này gồm constructor, getter và toString().

```
    // MenuItem.java
public class MenuItem {
   private String name;
   private double price;

   public MenuItem(String name, double price) {
      this.name = name;
      this.price = price;
   }

   public String getName() {
      return name;
   }

   public double getPrice() {
      return price;
   }

   @Override
   public String toString() {
      return "MenuItem [name=" + name + ", price=" + price + "]";
   }
}
```

Các lớp menu ban đầu: Lớp menu BreakfastMenu:

- Bên trong lớp này có chứa một mảng kiểu MenuItem dùng để lưu trữ các item của menu. Ta tạo một biến static lưu giá trị độ dài tối đa của mảng này. Biến numberOfItems là index của mảng.
- Trong hàm dựng, ta sẽ thêm giá trị vào mảng thông qua phương thức addItem(). Phương thức addItem() sử dụng vòng for để thêm các phần tử vào mảng, nếu phần tử thêm vào vượt quá độ dài mảng sẽ in ra thông báo.
- phương thức getMenuItems() sẽ trả lại mảng chứa các phần tử MenuItem, hay chính là thành phần của menu đồ ăn sáng (menu ngoài đời).

```
    // BreakfastMenu.java
public class BreakfastMenu {
   static final int MAX_ITEMS = 3;
     int numberOfItems = 0;
     MenuItem[] menuItems;
```

```
    public BreakfastMenu() {
      menuItems = new MenuItem[MAX_ITEMS];

      addItem("Noodle", 30000);
      addItem("Rib Congee", 25000);
      addItem("Hot Dog", 15000);
    }

    public void addItem(String name, double price) {
      MenuItem menuItem = new MenuItem(name, price);
      if (numberOfItems >= MAX_ITEMS) {
        System.err.println("Sorry, menu is full! Can't add item to menu");
      } else {
        menuItems[numberOfItems] = menuItem;
        numberOfItems = numberOfItems + 1;
      }
    }

    public MenuItem[] getMenuItems() {
      return menuItems;
    }
}
```

Lớp DinnerMenu ban đầu:

- Lớp này gần tương tự như lớp trên nhưng việc lưu trữ các phần tử lại được sử dụng ArrayList, nên ta không cần đặt giới hạn kích thước của nó.
- Phương thức getMenuItems() sẽ trả lại một List kiểu MenuItem chứa các phần tử đã được thêm vào, chúng là các thành phần trong menu đồ ăn tối (menu ở ngoài đời thực).

```
    // DinnerMenu.java
import java.util.ArrayList;
import java.util.List;

public class DinnerMenu {
   List<MenuItem> menuItems;

   public DinnerMenu() {
      menuItems = new ArrayList<MenuItem>();
```

```
        addItem("Beefsteak", 100000);
        addItem("Seafood", 200000);
        addItem("Hotpot", 300000);
    }

    public void addItem(String name, double price) {
        MenuItem menuItem = new MenuItem(name, price);
        menuItems.add(menuItem);
    }

    public List<MenuItem> getMenuItems() {
        return menuItems;
    }
    // other menu methods here
}
```

Bây giờ, chúng ta sẽ bắt tay vào thiết kế các lớp Iterator và các lớp con của nó. Các lớp con này sẽ tương ứng với các menu ở trên, Ta sẽ tạo ra BreakfastMenuIterator và DinnerMenuIterator.

```
    // Iterator.java
public interface Iterator {
    boolean hasNext();

    MenuItem next();
}
```

Vì ở bên trên, ta thấy BreakfastMenu sử dụng mảng để lưu trữ các menuItems, vì vậy trong lớp BreakfastMenuInterator này, ta sẽ tạo ra một thuộc tính là mảng có kiểu MenuItem, biến int position sẽ là chỉ số index trong mảng. Constructor của lớp này sẽ cho phép truyền vào một mảng kiểu MenuItem. Phương thức hasNext() sẽ kiểm tra xem mảng đã hết phần tử chưa bằng cách so sánh độ dài mảng với position. next() sẽ trả về phần tử tiếp theo trong mảng (có kiểu MenuItem) nếu hasNext() trả ra "true".

```
    // BreakfastMenuIterator.java
public class BreakfastMenuIterator implements Iterator {
    MenuItem[] items;
    int position = 0;

    public BreakfastMenuIterator(MenuItem[] items) {
        this.items = items;
    }
```

```
    public MenuItem next() {
        return items[position++];
    }

    public boolean hasNext() {
        return items.length > position;
    }
}
```

Lớp DinnerMenuIterator cũng tương tự như BreakfastMenuIterator, chỉ khác là ta sẽ tạo ra thuộc tính đối tượng là một ArrayList kiểu MenuItem.

```
    // DinnerMenuIterator.java
import java.util.List;

public class DinnerMenuIterator implements Iterator {
    List<MenuItem> items;
    int position = 0;

    public DinnerMenuIterator(List<MenuItem> items) {
        this.items = items;
    }

    public MenuItem next() {
        return items.get(position++);
    }

    public boolean hasNext() {
        return items.size() > position;
    }
}
```

Chúng ta cũng cần thiết kế lại các lớp BreakfastMenu và DinnerMenu. Ta sẽ tạo ra một interface Menu là superclass của 2 lớp này, interface Menu sẽ khai báo một phương thức createIterator(), ta cần thêm phương thức này vào 2 lớp BreakfastMenu và DinnerMenu.

```
    // Menu.java
public interface Menu {
    Iterator createIterator();
}
```

Thay đổi ở lớp BreakfastMenu: thêm từ khóa "implements" và phương thức createItera-

tor(). Phương thức này sẽ tạo ra Iterator tương ứng với menu bằng cách dùng hàm dựng của các lớp ConcreteIterator.

```
    // BreakfastMenu.java
public class BreakfastMenu implements Menu {
       ...
       // All code still the same
     public Iterator createIterator() {
       return new BreakfastMenuIterator(menuItems);
     }
}
```

Tương tự, ta cũng thiết kế lại lớp DinnerMenu:

```
    // DinnerMenu.java
public class DinnerMenu implements Menu {
       ...
       // All code still the same
   public Iterator createIterator() {
      return new DinnerMenuIterator(menuItems);
   }
}
```

Như vậy, ta đã tạo xong các ConcreteIterator và các ConcreteAggregate theo sơ đồ. Giờ đây, ta sẽ thiết kế thêm một lớp có tên Waitrees chứa các phương thức in ra menu là hoàn thành chương trình.

```
    // Waitress.java
public class Waitress {
   Menu breakfastMenu;
   Menu dinnerMenu;

   public Waitress(Menu breakfastMenu, Menu dinnerMenu) {
      this.breakfastMenu = breakfastMenu;
      this.dinnerMenu = dinnerMenu;
   }

   public void printMenu() {
      Iterator breakfastIterator = breakfastMenu.createIterator();
      Iterator dinnerIterator = dinnerMenu.createIterator();

      System.out.println("MENU\n----\nBREAKFAST");
      printMenu(breakfastIterator);
```

```
        System.out.println("\nDINNER");
        printMenu(dinnerIterator);

    }

    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
        }
    }
}
```

Trong lớp này, chúng ta sử dụng kỹ thuật compositon và delegation, lớp Waitress sẽ chứa 2 tham chiếu Menu. Ta có thể có 2 phương thức printMenu() và printMenu(Iterator iterator) dựa vào tính đa hình tĩnh (overload) trong lập trình hướng đối tượng.

- Phương thức printMenu(Iterator iterator) cho phép truyền vào một đối số có kiểu Iterator, nó sẽ sử dụng vòng for với điều kiện hasNext() trả lại "true" để thực hiện việc in các phần tử trong menuItem. Bằng phương thức next(), chúng ta có thể chuyển đến phần tử tiếp theo, cứ như vậy cho đến khi không còn phần tử nào nữa.
- Phương thức printMenu(): phương thức này sẽ tạo ra 2 tham chiếu kiểu Iterator mới bằng việc ủy quyền cho các tham chiếu Menu sử dụng createIterator() của chúng. Bên trong phương thức này sẽ gọi tới phương thức printMenu(Iterator iterator) ở dưới để in ra menu.

Lớp test:

```
    // Test.java
public class Test {
    public static void main(String args[]) {
        Menu breakfastMenu = new BreakfastMenu();
        Menu dinnerMenu = new DinnerMenu();
        Waitress waitress = new Waitress(breakfastMenu, dinnerMenu);
        waitress.printMenu();
    }

}
```

Output:

```
MENU
```

```
----
BREAKFAST
Noodle, 30000.0 -- Rib Congee, 25000.0 -- Hot Dog, 15000.0 --
DINNER
Beefsteak, 100000.0 -- Seafood, 200000.0 -- Hotpot, 300000.0 --
```

Như vậy, ta đã có thể in được các phần tử chứa ở các tập hợp khác nhau (ArrayList và Array) bằng việc sử dụng Iterator Pattern. Chúng ta có thể giải quyết bài toán này đơn giản hơn bằng cách sử dụng thư viện java.util.Iterator của java, thư viện sẽ giúp chúng ta giải quyết việc sử dụng Iterator Pattern nhanh hơn rất nhiều. Tuy vậy, việc cùng nhau xây dựng lại pattern này cũng giúp ta hiểu hơn về cách vận hành của nó.

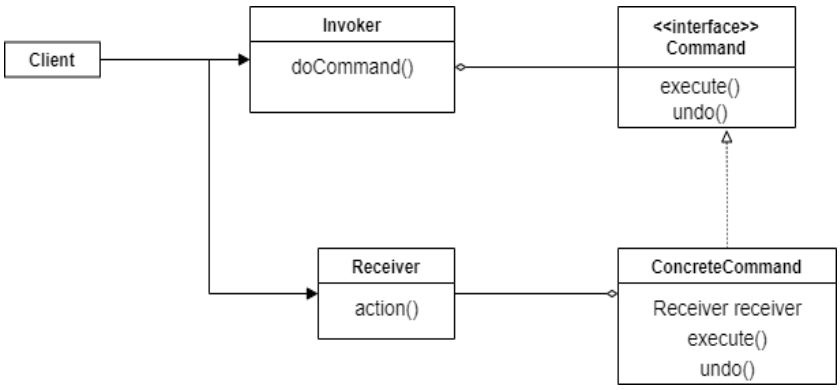
Ứng dụng trong thực tế của Iterator pattern: Chúng ta có thể tìm thấy pattern này trong thư viện java.util.Iterator và trong java.util.Enumeration.
Bấm vào đây để trở lại mục lục.

3.4 Command Pattern

Command Pattern là gì? Command Pattern là một pattern thuộc nhóm hành vi, cho phép đóng gói mọt request (yêu cầu) dưới dạng đối tượng, qua đó ta có thể tham số hóa đến nhiều yêu cầu khác nhau như queue, log, hỗ trợ các phép toán undoable như undo, redo.
Khi Client gửi đi request, request này sẽ được đóng gói dưới dạng một đối tượng gọi là Command Object, command object sẽ gọi đến các đối tượng thực thi cụ thể để thực hiện câu lệnh.

Mục đích sử dụng Command Pattern: Pattern này thường được sử dụng khi muốn lưu lại thông tin về trạng thái hiện tại cũng như các câu lệnh mà một đối tượng thực hiện, thuận tiện cho việc thực hiện các tác vụ như undo, callback (các tác vụ cho phép thực hiện lại). Pattern này cũng giúp che giấu cách thực hiện các request với Client.

Sơ đồ thực hiện của Command Pattern: Sơ đồ cấu trúc của Command Pattern có thể được biểu diễn như sau:



Trong đó:

- Client chịu trách nghiệm cho việc tạo các ConcreteCommand và cài đặt các Receiver tương ứng với nó.
- Invoker sẽ nhận lệnh từ Client, nó sẽ tiếp nhận các ConcreteCommand và sau đó gọi phương thức doCommand() của mình. Invoker sẽ ủy quyền (delegate) cho một đối tượng có kiểu Commmand để thực hiện các phương thức bên trong doCommand().
- Lớp Command sẽ được thiết kế là một interface (code to an interface) nhằm tạo ra một kiểu supertype cho các ConcreteCommand bên dưới. Interface này sẽ khai báo 2 phương thức là execute() và undo().
- Các lớp ConcreteCommand imlements từ Command sẽ thực hiện execute() và undo() theo như ràng buộc của chúng. Tuy nhiên không các phương thức này không phải do trực tiếp ConcreteCommand thực hiện. Các lớp này sẽ giữ một tham chiếu kiểu Receiver bên trong (composition) và sử dụng tham chiếu này để thực hiện execute() hay undo() bằng cách dùng "receiver.action()".
- Phương thức undo() là một phần rất thủ vị trong pattern này. Nó sẽ thực hiện một hành động có logic ngược với execute() ngay trước đó, chúng ta sẽ hiểu rõ điều này qua ví dụ minh họa.
- Receiver sẽ là lớp chứa các phương thức thật sự để thực hiện các request.

Code minh họa Command Pattern: Ta sẽ code minh họa Command Pattern thông qua việc thiết kế chương trình công tắc đèn đơn giản. Công tắc đèn

- Ta sẽ thiết kế Invoker lúc này sẽ là một lớp Remote, cho phép ta chọn nút bật đèn hay tắt đèn.
- interface Command như trên, khai báo 2 phương thức execute() và undo()
- Tiếp theo là các ConcreteCommand implements lại Command và thực hiện các ràng buộc về phương thức.

- Receiver ở đây sẽ là cây đèn (Light), nó mới là đối tượng thực sự bật hay tắt.

Chương trình cụ thể: Đầu tiên ta sẽ có một lớp Light đơn giản, lớp này chỉ gồm các phương thức như turnOn() và turnOff():

```
    // Light.java
public class Light {
   public void turnOn() {
      System.out.println("ON");
   }

   public void turnOff() {
      System.out.println("OFF");
   }
}
```

Tiếp theo là interface Command khai báo 2 phương thức execute() và undo(), 2 lớp TurnOnCommand và TurnOffCommand implements từ nó.

```
    // Command.java
public interface Command {
   void execute();

   void undo();
}
```

Trước khi đi vào phần code của 2 lớp TurnOnCommand và TurnOffCommand, chúng ta cần hiểu rõ cách thức hoạt động của execute() và undo(). Đối với lớp TurnOnCommand, ý nghĩa của phương thức execute() của nó sẽ là thực hiện hành động bật đèn và ngược lại của bật đèn sẽ là tắt đèn. Ta sẽ thiết kế phương thức undo() sao cho hành động mà nó thực hiện ngược lại (về mặt logic) với hành động được thực hiện trong execute().
Điều đó được thực hiện trong code như sau:

```
    // TurnOnCommand.java
public class TurnOnCommand implements Command {
   Light light;

   public TurnOnCommand(Light light) {
      this.light = light;
   }

   public void execute() {
      light.turnOn();
   }
```

```
    public void undo() {
        light.turnOff();
    }
}

    // TurnOffCommand.java
public class TurnOffCommand implements Command {
    Light light;

    public TurnOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOff();
    }

    public void undo() {
        light.turnOn();
    }
}
```

Như ta thấy, các lớp này chứa một tham chiếu Light bên trong và ủy quyền cho nó thực hiện execute() hay undo(). Như vậy là đã xong phần Command và Receiver, cuối cùng ta chỉ cần thiết kế phần Invoker (sẽ là Remote) và Client là hoàn thành chương trình.

Phần Remote sẽ được thiết kế giống như một chiếc điều khiển: bấm nút bật để bật đèn, bấm nút tắt để tắt đèn, bấm nút undo để quay lại trạng thái ngay trước nó.

- Hàm constructor sẽ cho phép ta khởi tạo một đối tượng Remote bằng cách truyền vào hai đối số kiểu Command là supertype của TurnOnCommand và TurnOffCommand.
- Ta sẽ đặt các tham chiếu Command trong lớp Remote này, và ủy quyền cho chúng thực hiện các phương thức bật hay tắt đèn tương ứng bằng cách sử dụng các execute() đã được ghi đè ở TurnOnCommand hay TurnOffCommand.
- Thuộc tính undoCommand sẽ cho phép ta lưu trạng thái hiện tại của đối tượng. Ta sẽ làm điều này bằng cách sau mỗi khi thực hiện các lệnh bật đèn hay tắt đèn, ta sẽ gán undoCommand "=" với tham chiếu Command vừa thực hiện nhiệm vụ đó.

```
    // Remote.java
public class Remote {
    Command turnOn;
```

```
    Command turnOff;
    Command undoCommand;

    public Remote(Command turnOn, Command turnOff) {
        this.turnOn = turnOn;
        this.turnOff = turnOff;
    }

    public void pressTurnOnButton() {
        System.out.println("Turn the Light on");
        turnOn.execute();
        undoCommand = turnOn;
    }

    public void pressTurnOffButton() {
        System.out.println("Turn the Light off");
        turnOff.execute();
        undoCommand = turnOff;
    }

    public void undo() {
        System.out.println("Undo");
        undoCommand.undo();
    }
}
```

Lớp test:

```
    // Client.java
public class Client {
    public static void main(String[] args) {
        // create Receiver
        Light light = new Light();

        // create ConcreteCommand
        Command turnOn = new TurnOnCommand(light);
        Command turnOff = new TurnOffCommand(light);

        Remote remote = new Remote(turnOn, turnOff);

        remote.pressTurnOnButton();
        remote.pressTurnOffButton();
```

```
        remote.undo();
    }
}
```

Output:

```
Turn the Light on
ON
Turn the Light off
OFF
Undo
ON
```

Ứng dụng Command Pattern trong thực tế: Pattern này được sử dụng rất nhiều trong việc thiết kế các ứng dụng, phần mềm, đặc biệt là những ứng dụng có tính năng undo (cho phép người dùng quay lại bước trước).
Bấm vào đây để trở lại mục lục.

IV Tài liệu tham khảo

Các tài liệu tham khảo chính:

- Slides bài giảng OOP - Design Patterns - Giảng viên Quản Thái Hà - Đại học Khoa học Tự nhiên, Đại học Quốc gia Hà Nội.
- Sách Head First Java 2nd Edition - tác giả Kathy Sierra và Bert Bates.
- Bản dịch tiếng việt cuốn Head First Java
  toihocdesignpattern.com
- Series bài giảng về Design Pattern của GpCoder.
  https://gpcoder.com/4164-gioi-thieu-design-patterns/
- Series Video Youtube về Design Pattern - Derek Banas.
  https://www.youtube.com/channel/UCwRXb5dUK4cvsHbx-rGzSgw

Em xin gửi lời cảm ơn chân thành đến Khoa Toán - Cơ -Tin học trường Đại học Khoa học Tự nhiên, ĐHQG HN đã tạo điều kiện cho chúng em học tập và nghiên cứu môn học này, đặc biệt là thầy giáo Quản Thái Hà - giảng viên chính của bộ môn Các thành phần phần mềm đã tận tình hướng dẫn chúng em trong suốt thời gian qua.

Các phần định nghĩa và kiến thức trong bài tiểu luận được em tham khảo, dịch và diễn giải lại từ sách Head First Java và những trang web uy tín trên internet, tuy nhiên do kiến thức và kinh nghiệm vẫn còn hạn chế nên không thể tránh khỏi việc tồn đọng một

số thiếu sót. Kính mong thầy xem xétvà góp ý giúp bài tiểu luận của em trở nên hoàn thiện hơn. Em xin chân thành cảm ơn thầy!